DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DAVID MANUEL MARQUES ANTUNES
BSc in Computer Science

TOWARDS GENERIC AND SCALABLE NETWORK EMULATION

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

TOWARDS GENERIC AND SCALABLE NETWORK EMULATION

DAVID MANUEL MARQUES ANTUNES

BSc in Computer Science

Adviser: João Carlos Antunes Leitão
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: João Ricardo Viegas da Costa Seco
Associate Professor, NOVA University Lisbon

Rapporteur: Hugo Alexandre Tavares Miranda
Associate Professor, University of Lisbon

Adviser: João Carlos Antunes Leitão
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2025

Towards Generic and Scalable Network Emulation

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [1].

Acknowledgements

I would like to begin by thanking my advisor, Professor João Leitão. His support, guidance, knowledge and friendship were crucial for the completion of this dissertation.

To all the people that make up Nova School of Science and Technology, and in particular the Department of Computer Science, a heartfelt thanks for making this a place where I was able to grow as a person and to learn much of what I know today.

I must thank my family for their support.

To all those with whom I worked during my time here, but especially, those at our work room 248, thank you for being a part of this journey and making it a more fulfilling one.

A special acknowledgement goes out to my friends, João Bordalo, for being my unofficial advisor, helping me to review some of my work, Diogo Paulico, for allowing me to bounce ideas off of him, and Maria Eduarda, Jacinta Sousa, Katarina Dyrebi, Tiago Teles, André Palma for making this journey a much more enjoyable one. Friends are the family we get to choose, and I hope we continue to be family for many years to come.

Abstract

With the rise in the adoption of microservice architectures, distributed systems, and cloud in recent years, the reliability of network infrastructure has become increasingly critical. In distributed systems, proper network management is essential to ensure smooth operation and quick response times for clients. However, as systems grow more complex, integrating new features can become challenging, potentially leading to system failures, service disruptions or performance degradation. To mitigate such risks, a robust testing environment is necessary to ensure that new features do not compromise the system's stability and functionality.

Solutions involving physical testbeds become impractical when dealing with distributed systems consisting of hundreds or even thousands of nodes, as these require vast computational resources and cumbersome configuration. Cloud technologies which allow for simplified deployment of many nodes on a global scale could be leveraged to test such systems, however their adoption for this purpose is hindered by the high operating costs. Simulation and Emulation tools help create a less complex testing environment, while replicating the complexity of real-world execution environments and producing accurate results. Nonetheless, currently available tools are limited when it comes to dynamic control of the experiment at runtime.

In this work we introduced a tool designed to test distributed systems, leveraging XDP and eBPF for traffic shaping and Docker for its containerization technologies for network emulation, and evaluated it against the current state of the art. The main contribution of this work is a network emulator–GONE–which facilitates testing and debugging distributed applications through real-time configuration of the network model and its properties. Through experimental evaluation, we found that the performance is comparable to the current state of the art, while allowing for dynamic control at runtime.

Keywords: Distributed Systems, Emulation, Networking

Resumo

Com o aumento da adoção de arquitetura de microsserviços, sistemas distribuídos e cloud a confiabilidade das infraestruturas de rede ganhou uma importância reforçada. Em sistemas distribuídos gestão apropriada da rede é essencial para garantir um bom funcionamento do sistema e tempos de resposta baixos para os clientes. Ao mesmo tempo, estes sistemas têm ficado mais complexos, e a integração de novas funcionalidades pode levar a falhas de sistema, interrupção de serviço ou degradação do desempenho. De forma a mitigar estes riscos, é essencial conceber um ambiente de testes robusto.

Soluções que dependem de infraestruturas físicas de teste são pouco práticas quando estamos a lidar com sistemas distribuídos que dependem de centenas ou milhares de nós, visto que requerem elevados recursos computacionais e configuração complicada. Tecnologias de cloud que permitem deployment de elevado número de nós, geograficamente distribuídos, poderiam ser instrumentados para testar estes sistemas, no entanto os seus custos de operações elevados inibem a sua adoção. Ferramentas de simulação e emulação permitem a criar um ambiente de teste menos complexo, replicando a complexidade de ambientes de execução reais e produzindo resultados precisos. No entanto, as ferramentas existentes, apresentam limitações ao nível do controlo dinâmico da experiência em tempo real.

Neste trabalho, introduzimos uma ferramenta desenhada para testar sistemas distribuídos que utiliza XDP e eBPF para alterar tráfego de rede e Docker devido à sua tecnologia de conteinerização que usamos para emulação de rede, esta ferramenta foi ainda testada com as soluções disponíveis atualmente. A principal contribuição deste trabalho é o emulador de rede GONE que facilita o teste de aplicações distribuídas ao permitir a configuração em tempo real do modelo da rede e das suas propriedades. A avaliação experimental demonstrou que o seu desempenho é equivalente ao das ferramentas existentes, permitindo, no entanto, controlo dinâmico da experiência.

Palavras-chave: Sistems Distribuídos, Emulação, Rede

Contents

List of Figures

List of Tables

List of Listings

Acronyms

RTT Round-Trip Time *(pp. 32, 47)*

SLOC Source Lines Of Code *(p. 32)*

TC Linux Traffic Control *(p. 41)*

UDP User Datagram Protocol *(p. 56)*

1 Introduction

In this chapter, we provide some context regarding this work that addresses the increase in the popularity of distributed systems, requiring new testing approaches, thus fueling the motivation for this work. We also present the objectives for this work and a description of our contributions.

1.1 Context

With the rise of microservice architectures, cloud adoption [2] in recent years, and specially with the emergence of web 3.0 and highly decentralized applications [3], the importance of network infrastructure becomes increasingly critical. In distributed systems, proper network management is essential to ensure smooth operation and quick response times for clients. However, as these systems grow more complex, integrating new features can become challenging, potentially leading to system failures or service disruptions. To mitigate such risks, a robust testing environment is necessary to test and ensure that new features do not compromise the system's stability and functionality.

To test distributed systems and evaluate their behavior, developers can resort to multiple approaches. One approach is leveraging existing testbeds, namely Emulab [4] or PlanetLab [5] (which has recently been discontinued), offering realistic deployment scenarios. These testbeds allow the experimenter to deploy their application in geo-distributed machines for testing. However, such platforms limit both the scalability of experiments since it is a service that is not exclusive to only one user, and is limited by the available infrastructure, in terms of configuration and its physical deployment.

Another avenue is Cloud solutions. Due to the growing popularity of cloud solutions [2], Cloud providers such as Azure [6], Google [7], and AWS [8] offer services to clients to rapidly deploy machines in any region worldwide to meet client demand. Similarly, developers can leverage these mechanisms to quickly deploy a testing environment to test and evaluate a new component or versions of the system. However, this scalability and flexibility comes with a monetary cost of deployment and limited control over the deployed infrastructure.

Another approach is to leverage simulation [9–11] or emulation [12–14] techniques. To avoid costs and configuring infrastructure, developers can rely on emulation or simulation tools for testing and validating distributed systems.

Simulation tools provide the developer full control over a network model, leading to deterministic experiments over a model or mock up of a distributed system. However, this deterministic approach leads to executing the distributed system in an isolated environment, and thus the observed results can differ from a real deployment.

An alternative to simulation is emulation. In this case, distributed applications execute in real time and interact with emulated components (typically the network), as if it were real systems. This approach provides similar results compared to real deployments, with the cost of higher processing power for emulating components, posing a problem when emulating large network topologies, as reaching hardware limits restricts testability.

1.2 Motivation

Due to new technological innovations, systems are increasing in complexity, often requiring inter-cooperation with other regional systems to ensure global coverage. Inter-cooperation between multiple systems leads to using coordination mechanisms to observe predictable behavior, ensuring correct operation. Proportionally, the increase in complexity forces the creation of better testing environments and more robust debugging tools, since in distributed systems, testing a particular component often requires the deployment of other services in the architecture. Furthermore, the network infrastructure has a huge impact in how communication between components occurs, and thus can change system operation, that can lead to bugs and hard to predict behavior.

As such, developers have many promising approaches for creating a testing environment. They can resort to utilizing existing testbeds, namely Emulab [4] or PlanetLab [5], or Cloud Providers, Azure [6], Google [7], and AWS [8]. However, the developer as to manually configure each machine, which is an arduous task and error-prone.

To avoid configuration overhead, it is possible to leverage simulation tools. Tools such as NS-3 [9] or OMNeT++ [10], provide a simulated network model, that can be leveraged to deploy an environment containing large network topologies, speeding up testing. Unfortunately, by resorting to simulation, the system will need to be executed on an isolated environment, and thus its observed behavior can differ when interacting with real systems.

An alternative is emulation. Existing emulation tools such as Kollaps [13] and Symphony [14] provide a scalable testing environment, capable of emulating large network topologies, facilitating the deployment of a distributed system across multiple machines, and providing mechanisms for network topology modification. However, due to its experiment based approach, all network changes must be configured before the experiment, limiting testability, since it is not possible to dynamically modify the network according to system behavior or other external factors.

In this work we have developed a scalable network emulator designed to test large-scale distributed systems, facilitating testing and debugging by utilizing a dynamic network topology approach with emulated network properties such as latency injection, bandwidth throttling, jitter and packet loss, and providing mechanisms to further expand its functionality, such as fine-grained latency control or emulating link-flapping.

1.3 Contributions

The main contributions of this work summarized as follows:

- A new network Emulator, GONE, that provides the user the ability to dynamically configure an emulated network topology through an existing API, allowing network modifications according to observable events.
- An experimental evaluation of GONE that demonstrates the application of desired emulation properties when compared to other network emulators, the capability for network changes, and the capability of adding new network behavior to an existing network link.

1.4 Document Structure

The remainder of this document is organized as follows:

Chapter 2 covers basic network concepts and presents the current limitations of existing solutions for testing and evaluating complex distributed systems. These limitations lead to a discussion of alternative techniques such as emulation and simulation, where their differences are discussed in a network context. Next, tools and techniques capable of interacting with relevant network properties will be described and discuss their relevancy to this work. Finally, there will be a discussion about related projects which have tried to simulate or emulate certain network infrastructures, describing the employed mechanisms to modify the network and their limitations.

Chapter 3 presents our contribution, GONE, detailing the requirements achieved with our work, followed by a description of the architecture. Following its architecture, it is presented its deployment steps, followed by its available API, providing a brief description of each endpoint. Finally, it is discussed its internal structures to successfully provide the observed emulation capabilities.

Chapter 4 describes how the evaluation of the contribution of this work is conducted. First, the testing goals of this work are defined across three distinct dimensions: network emulation, dynamic behavior, and extensibility. Next, it is presented the experiments conducted to evaluate network emulation, assessing how GONE applies

the desired network properties and compare the results obtained against the state-of-the-art emulator. Next, it is evaluated the stability of our work. Following network emulation, the evaluation of dynamic behavior is presented, demonstrating the mechanisms to affect the network topology during experiments, affecting application behavior. Lastly, extensibility of GONE is described, where examples are presented to demonstrate how the user can extend the functionality of GONE to create new network operations or obtain key insights during experiments.

Chapter 5 presents the main conclusion of this work and future work.

2

Related Work

In this chapter, fundamental concepts are presented throughout the sections, establishing the background for the solution proposed in Chapter 3. It first starts with a theoretical background regarding networking, followed by the analysis of relevant technologies, current approaches and their limitations, tools for manipulating network properties, and discussion about related work.

The structure of this chapter goes as follows: Section 2.1 introduces basic network concepts; Section 2.2 reflects current environments capable of deploying large network topologies and their limitations; Section 2.3 introduces the concepts of simulation and emulation and discusses their application in a network context; Section 2.4 presents network management and manipulation tools related to this work; Section 2.5 introduces and reviews concrete network simulators; Finally, Section 2.6 describes and discusses existing network emulators and how they fail to meet our defined goals.

2.1 Structure of a Network

Before digging further into subjects related to networking, it is important to first introduce basic network concepts to aid in understanding the remainder of this document. This prompts the question: what constitutes a network? A network comprises a set of machines connected to one another, capable of communicating through the exchange of messages. However, a network is not as simple as this basic description suggests; it involves complex mechanisms such as routing protocols, (e.g BGP and OSPF to name a few), that determine paths for data transmission.

For communication to occur, there must exist a physical connection, known as data links, enabling the flow of information. Having a physical connection to every machine is impractical, specialized hardware devices capable of interconnecting multiple systems such as routers, switches, and hubs are employed to connect multiple systems in the network. This approach leads to the creation of large networks requiring special communication protocols to interconnect and coordinate, forming the Internet known as of today.

To properly distinguish between different network mechanisms, researchers resort to the OSI Model to provide a clear distinction of network components. According to the OSI Model, a network can be structured in seven layers, as shown in Figure 2.1.

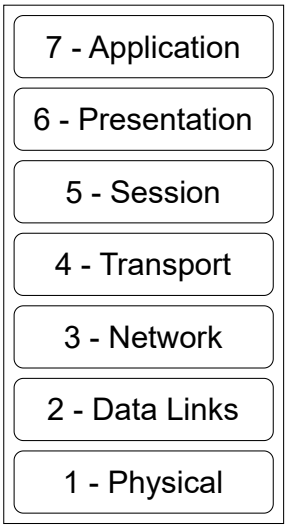


Figure 2.1: Seven Layer of Networking according to the OSI Model

- **Layer 1 (Physical Layer):** represents the physical connection between devices, enabling the transmission of data.
- **Layer 2 (Data Link Layer):** Handles direct network connections between two devices, communicating through Ethernet frames.
- **Layer 3 (Network Layer):** Manages the exchange of information via network packets, allowing communication without direct connection to the destination.
- **Layer 4 (Transport Layer):** Ensures the reliable flow of information between two endpoints.
- **Layer 5, 6, 7 (Session, Presentation, and Application Layer):** Provides structures required for the proper functioning of applications, including session management, data representation, and application-specific protocols.

The primary focus of this work will be on the first three layers of the OSI Model, regarding the understanding, deployment, and network management of network devices.

Throughout this document, routers and switches belonging to a network will be referred to as network devices or network nodes; the physical connection between two network devices as a link or data link; the physical arrangement of network devices and data links as a network topology; computers or running applications in the network as hosts or end hosts.

As previously mentioned, a small network can be composed of a large amount of network devices, becoming impossible for researchers to build their own physical testbeds,

requiring manual configuration for a particular network topology and execute experiments. Alternative approaches researchers can resort to mitigate such impossibility will be discussed in the following sections.

2.2 Cloud and Physical Testbeds

As systems grow in complexity, it becomes increasingly difficult to deploy them on a single machine due to resource contention, degrading system throughput.

As a countermeasure to this limitation, developers will build their systems to work across multiple machines, allowing the balance of load more effectively, as well as meet client demand. However, this strategy creates more complex system, requiring coordination mechanisms to ensure correct operation [15].

Testing and evaluating simple distributed systems on a local physical testbed with a few machines is feasible, cost-effective and straightforward. However, challenges arise as these systems are deployed in multiple regions, requiring coordination.

One approach to this issue is by resorting to research testbeds, i.e., Emulab [4] and PlanetLab [5], allowing the deployment of hundreds of machines across multiple regions. These platforms allow researchers to request machines, providing an environment capable of configuring desired network topologies and conditions. However, these testbeds are shared by a community of researchers, resulting in restricted availability, and users are responsible for configuring the testbed to meet their specific requirements for testing and validating large distributed systems.

Another approach is leveraging cloud infrastructure. Cloud providers such as Google [7], AWS [8] and Microsoft Azure [6] are amongst the most popular cloud providers currently, operating on a global level [16], with data centers dispersed worldwide, offering tools for provisioning, networking, and automation. These services however, offer with simple and straightforward interfaces to ease deployments for clients, abstracting configuration. This results in limited observability about the deployed infrastructure. This ease of use interface comes with a cost associated, where creating a testbed composed of servers deployed in multiple regions results in high operational costs, making it financially unsustainable in the long term.

While traditional testbeds and cloud infrastructure have their uses, researchers often seek more affordable and flexible alternatives. Techniques such as simulation [9, 10] and emulation [13, 17, 18] are great cost-effective approaches to model complex systems under specific conditions, maintaining accurate results when compared to real deployments. These approaches will be discussed in later sections of this chapter.

2.3 Simulation and Emulation in a network context

2.3.1 Simulation

In Computer Science, researchers resort to simulation to test the behavior of a particular component, a set of components or an entire system given certain conditions [19]. The creation of an accurate model built in a simulation environment allows engineers the ability to have full control over the experiment, since every relevant metric must be modeled correctly, without requiring the complete model of the system. Accurate simulation models can be used alongside real hardware, providing insights into system functionality at an early stage [20, 21].

The configuration of relevant metrics regarding the behavior of a model provides researchers a deeper understanding of its execution. In a simulation environment, the practitioner has full control over the models' inputs and thus can control how the experiment behaves, such as time itself. This control results in the capability of deploying large and complex systems in slower hardware. By utilizing simulated time, the simulator obtains execution time to calculate correct behavior.

Unfortunately, there are some drawbacks. The ability to execute on simulated time is good for testing large and complex systems, allowing correct behavior, but inversely, increases the duration of the experiment, leading to evaluation times so long that it becomes impractical. The usage of simulated time also results in the simulated environment be disconnected from real-world events. Furthermore, the execution of the model in a simulated fashion could lead to inaccurate results when compared to its implementation in a real testing environment [22] due to external anomalies not accounted for in the simulated model.

2.3.2 Emulation

Another approach mentioned in Section 2.2 to test distributed systems is to resort to emulation [23]. Emulation, as in simulation, requires the construction of a model.

Emulation tries to design a model to replicate its physical counterpart. Unlike simulation, which focuses on abstract representations and having full control over the system it wishes to test, emulation strives to mimic the behavior and characteristics of a component. The advantage of this approach is the ability to perform live testing, allowing researchers to test the emulated model alongside real and verified systems. This often leads to more accurate results compared to physical testbeds, as it accounts for unpredictable behavior from external inputs [24].

However, emulation also has its restrictions. To work alongside real systems, the model cannot be executed on simulated time, thus constraining its execution to the physical capabilities of the hardware. Gouveia et al. [13] demonstrated these limitations while evaluating their scalable network emulator, Kollaps. In their testing, they compare

the accuracy of handling large network topologies of different network emulators against theirs, demonstrating the importance of scalability and precision in emulation systems.

2.3.3 Network Context

In networking, a wide range of systems can be tested. Systems can range from a simple simulation of a communication protocol like TCP [25], or emulating datacenter networks [12].

To create testing environments to produce accurate results without being computationally expensive, we arrive at a system which meshes the concept of emulation with simulation, as discussed by the authors in [23]. Their study highlights that network emulation is the bridge between the usage of simulation and live environments. The correct approach to solve the problem is by defining simpler but accurate network models, capable of interacting with real technology. Over the years, emerged network simulation tools such as NS-3 [9] or OMNeT++ [10], have been developed but ultimately are limited to the execution of the network model. Section 2.5 elaborates further on how such simulators operate.

Work has also been done to integrate simulators closer to a live testing environment. Systems such as Dockemu [26] which resorts to tools like Docker [27] to build a containerized environment, allowing the experimenter to run arbitrary code while also modifying the Docker network configuration to connect containers with the simulation running in NS-3 [9]. By combining real applications with simulations the simulated model from NS-3 is executed in real-time, bringing simulations closer to real-world scenarios.

As noted by Lochin et al. [23], constructing simple network models suited to interact with real world events is crucial for network emulation, capable of generating accurate results, when modeling various components of the network. This simplification reduces computational requirements by avoiding the full emulation of the system. Examples of network emulators adopting this approach include NetEm [28], Mininet [12], Maxinet [29], and Kollaps [13], to name a few. Section 2.6 provides more details on the operation of the mentioned emulators.

2.4 Network Management and Manipulation Tools

Before diving into the discussion of existing network simulators and emulators, it is important to first discuss existing tools present in the Linux ecosystem capable of modifying the network, providing the groundwork for the contribution of this work.

The following sections start off by describing eXpress Data Path (XDP), a high-performance packet processing framework in Section 2.4.2; Section 2.4.1 presents extended Berkeley Packet Filter (eBPF), an instruction set and virtualized environment capable of executing in XDP; Section 2.4.3 presents Linux Traffic Control (TC), a powerful network tool capable of modifying the behavior of network interfaces; Section 2.4.4 introduces

Netfilter, a network tool capable of applying packet shaping to incoming packets; Section 2.4.5 discusses a different approach to configure a network, by resorting to Software Defined Networks (SDN); Section 2.4.6 discusses approaches to integrate unmodified application code; Section 2.4.7, addresses scalability tools for the proposed solution; Finally, in Section 2.4.8 reflects how these tools and techniques are helpful in contributing to the development of a solution as the one proposed in this dissertation.

2.4.1 Extended Berkeley Packet Filter

The Extended Berkeley Packet Filter [30] (eBPF) is the successor of the Berkeley Packet Filter [31] (BPF). Its provides an instruction set and an execution environment within the Linux Kernel, enabling the modification of packet processing in network devices. eBPF programs are compiled from restricted C language and then executed in kernel-space, through a virtual machine.

To perform network packet filtering, developers resorted to BPF. BPF provided a set of instructions to be executed on a virtual machine in the kernel. The number of instructions available allow the validation of code to ensure security and avoiding kernel crashes. Tools such as tcpdump [32] use BPF to create the desired filter for network observability.

eBPF extends BPF with additional instructions expanding available memory. The new instructions allow making function calls and modifying data through data structures known as maps. Maps are key-values stores capable of storing user-defined data. User-space processes can create multiple maps that are accessible to both other user-space processes and eBPF programs executing in the kernel.

Another functionality of eBPF is the ability to call other eBPF programs, via tail calls [33], by reutilizing existing kernel memory to run the next ePBF program. This functionality simplifies the program design by creating modular and reusable code.

Several solutions have leveraged eBPF to reimplement or extend existing tools. For instance, the authors in [34] utilize eBPF to reimplement existing mechanisms provided by NetEm [28], and others used eBPF to enhance network monitoring [35] and observability [36, 37].

Unfortunately, eBPF programs come with limitations. As previously mentioned, eBPF only has access to a subset number of C language library functions. Another limitation is its limited stack space, with a maximum size of 512 bytes. Another restriction is the impossibility of executing loops, allowing for more complex programs to be defined.

eBPF can be utilized to further enhance the Linux network stack. Fig. 2.2 shows the mechanisms present in the Linux network stack. It is possible to attach eBPF programs to the Linux Traffic Control layer (see Section 2.4.3) or the eXpress Data Path, giving control of ingress traffic of a network interface to the user. The following section (Section 2.4.2) explains how to the XDP mechanism works, and how the user can leverage XDP and eBPF to interact with ingress traffic.

Figure 2.2: Layers of the Linux Network Stack

2.4.2 eXpress Data Path

The eXpress Data Path [38] (XDP), is a high-performance packet processing framework that operates at the kernel level.

By using eBPF alongside XDP, the user can integrate custom code to skip or work alongside the network stack, and can be integrated on the kernel, or directly executed on the network device. This approach allows the processing of network traffic even before the Linux kernel processes the incoming data, increasing performance of operations such as dropping or redirecting packets as demonstrated by the authors in [38]. Since XDP programs can be executed directly on the device driver, the host application is agnostic to changes made by XDP. Due to such operating approach, XDP programs avoids the overhead of context switching between user-space and kernel-space restrictions. Fig. 2.3 presents a diagram of the packet data flow of the typical Linux network stack or can be redirected directly to an application.

XDP is the first component to interact with a given network packet, allowing the configuration of redirection of the network traffic directly to an application through the AF_XDP [39] socket. There is a great code example of how to configure eBPF and the AF_XDP socket in the Github repository "xdp-tutorial" [40]. The annex I.1 provides an example of an eBPF program to redirect the network traffic to an AF_XDP Socket. The

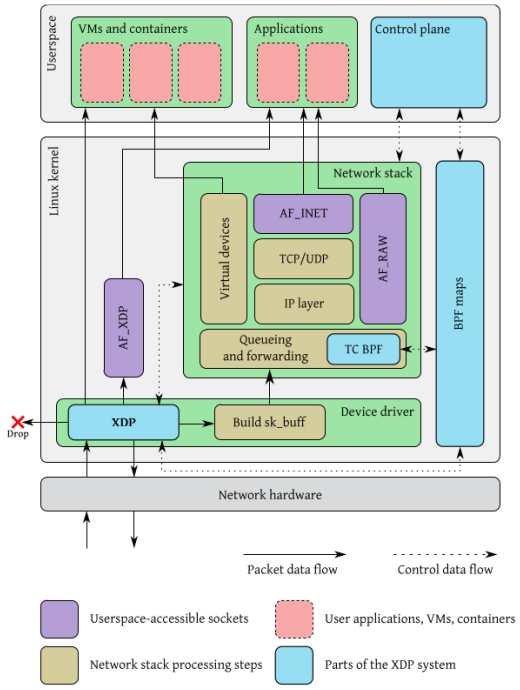


Figure 2.3: Data flow in the Linux Network Stack. Extracted from [38]

code snippet presents two possible paths for the network packet, where one is following the network stack path, by returning XDP_PASS, or be redirected to an AF_XDP socket, through `return bpf_redirect_map(&xsks_map, index, 0);`.

For the user-space application to access the network traffic, it requires the creation of a AF_XDP socket, and register its address to an eBPF map so that the eBPF program can correctly redirect the packet, as explained by Jonathan Corbet in [41]. The AF_XDP requires the association of an array of memory in user-space, for storing network frames. To manage this continuous block of memory to be able to receive or transmit network traffic, it is necessary to associate four queues, named "Fill Queue", "Transmit Queue", "Receive Queue", and "Completion Queue".

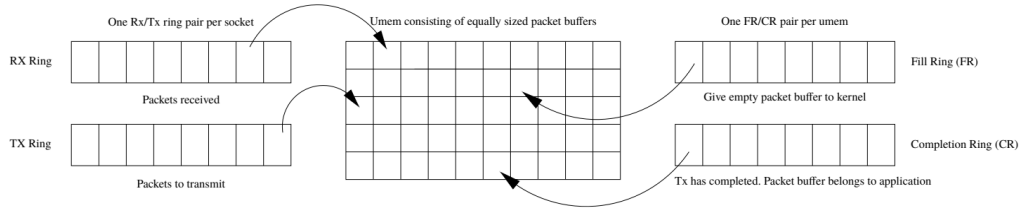


Figure 2.4: The four rings of AF_XDP and the UMEM containing all the packet buffers. Extracted from [39]

Figure 2.4 displays the typical logic to manage an AF_XDP socket. In this scenario, the "Receive Queue"and "Transmit Queue"are utilized by the application, while the "Fill Queue"and "Completion Queue", are managed by the kernel. In order to receive a network

frame, the user must submit the address of a frame of memory to the kernel. The kernel upon receiving the address, populates the frame with a new network packet, and signals the application through the "Fill Queue"the frame of memory contains new information. For transmission, the operation works in reverse. The application fills a memory frame with a network packet, and then passes the address to the kernel. After the kernel transmits the packet, it fills the "Completion Queue"to signal the user-space application the successful transmission.

This type of socket provides a framework to quickly receive or transmit network data, with the downside of losing all the existing network mechanisms present in the Linux Kernel, transferring that responsibility to the application.

2.4.3 Linux Traffic Control

The Linux Traffic Control [42–44] (TC) is a powerful network tool available on Linux systems. It enables the modification of the behavior of network traffic in egress. TC provides capabilities for shaping, scheduling, policing, and dropping network traffic, through mechanisms called *qdiscs*, short for "queuing disciplines", classes, filters.

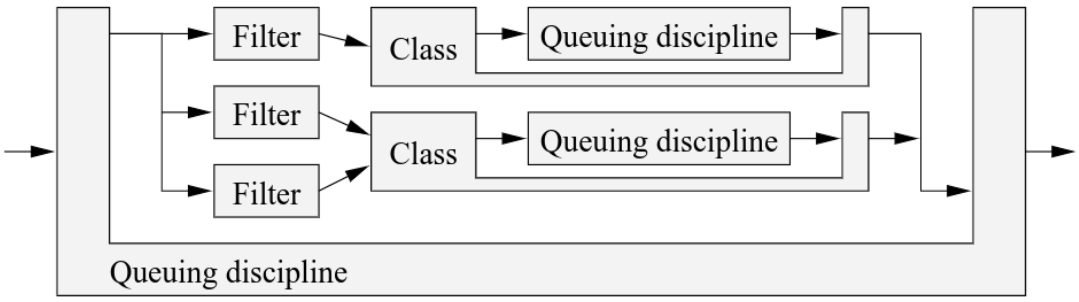


Figure 2.5: Simple queuing discipline with multiple classes. Extracted from [42]

Queues are the backbone of TC, storing incoming packets into a memory buffer, waiting processing. The behavior of a queue is defined by the chosen algorithm. TC distinguishes a *qdisc* between its classful or classless property. Classful *qdiscs* can contain classes, and filters. Classes allow the classification of traffic, providing the capability of configuring new queue disciplines inside a *classful qdisc*. Filters on the other hand, are utilized to filter traffic in classful *qdiscs* to other *qdiscs*. Fig. 2.5 shows a simple TC configuration pattern. TC provides different algorithms with configurable parameters, some examples are:

- **First-In-First-Out (FIFO)**: processes packets in the same order of arrival. The queue size can be limited, causing packets to drop.
- **Token Bucket Filter (TBF)**: generates tokens at a predefined rate, limiting the number of packets processed. When a token is generated, a bucket of packets is processed.

- **Priority Scheduler (PRIO)**: retrieves packets according to class order. Fetches packets from the next class only if the previous class is empty.

Classless *qdiscs* on the other hand, represent the last queuing disciplines to configure in traffic shaping, as these *qdiscs* do not allow adding new classes or filters. Some examples of classless *qdiscs* are:

- **Choke (CHOK)**: queuing discipline that manages network traffic according to bandwidth usage, throttling connections that heavily uses the queue.
- **Stochastic Fair Blue (SFB)**: manages the queue according to congestion based on packet loss and link usage.
- **Stochastic Fairness Queue (SFQ)**: reorders packets in the queue to provide progress for each connection present.

Filters can be utilized to apply packet classification, and policing. Packet classification allows marking packets for redirection to other queues. The classification can occur during the arrival, routing or transmitting of a packet.

Policing operates akin to an "if statement". For instance, traffic can be limited, with the condition of defining an upper limit on the current bandwidth threshold. If this policy is violated, packets are dropped, else they continue.

Despite its flexible capabilities, TC has some limitations. Its syntax is complex, making configuration challenging. TC was designed to mangle egress traffic, but is limited in its functionality for managing ingress. Another downside is the design of TC, which presupposes that the network traffic is processed in only one direction, thus limiting the capability of reutilizing existing structures to process network traffic. Lastly, for each queuing discipline, TC will execute every attached filter trying to find a match to correctly redirect the network packet. This approach leads to high processing time for each packet in case of a *qdisc* containing hundreds of filters.

2.4.4 Netfilter

Going up in the network stack of Linux, Netfilter [45, 46], a packet manipulation framework, allows the execution of callback functions to configured network events. Programs such as Arptables [47], Iptables [48] and Ip6tables [49] and recently Nftables [50–52] leverage Netfilter to manipulate packets in their respective network layer. Arptables manipulates packets at the Second Layer of the OSI Model (see Section2.1), handling Ethernet Address Resolution Protocol [53] (ARP) communication while Iptables and Ip6tables manages IPv4 [54] and IPv6 [55] communication, respectively.

Figure 2.6 shows the available hooks to configure network rules to mangle with network traffic. Netfilter allows the aforementioned programs to process packets, apply filters, forward packets, and do Network Address Translation (NAT) through the available

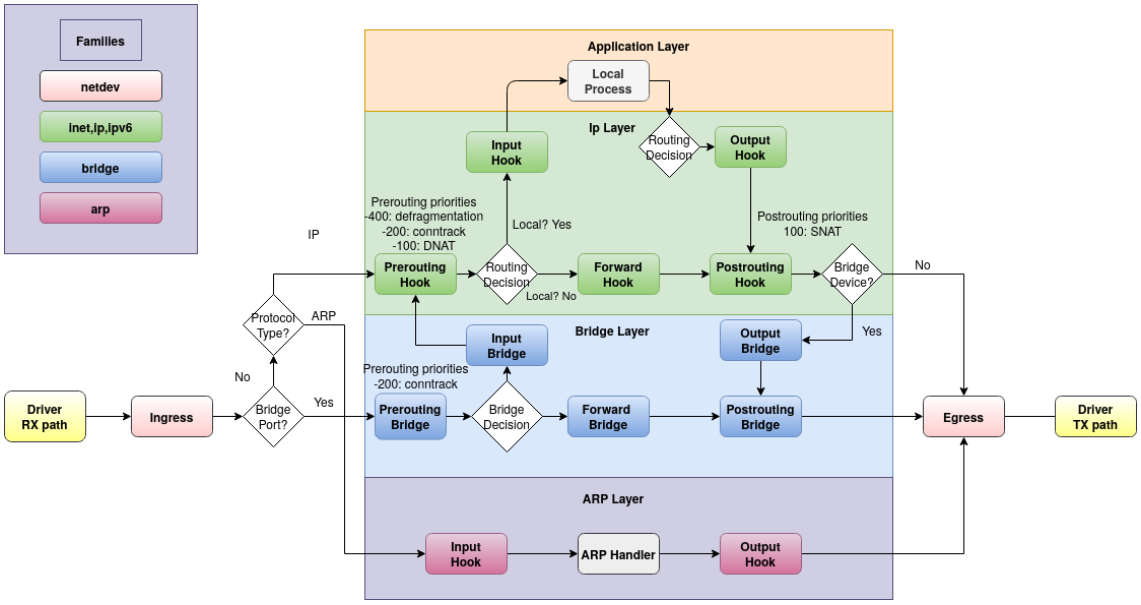


Figure 2.6: Hooks available in the Linux network stack Nftables can configure rules. Extracted from [52]

network hooks. Those hooks are executed before the packets are passed to the TC layer of the Linux Kernel. This implementation permits hiding network devices behind a particular machine, thus reducing the number of public IPs [56]. Netfilter also allows the ability to redirect network packets to userspace through a *nfqueue*, allowing an application to mangle with network traffic. In this scenario, the Linux Kernel gives control over the network traffic to the application and awaits for a reply from the program.

The program, when connected to a nfqueue, receives network traffic from the netfilter layer, passing the ownership to the application. The application is now responsible to signal to process the received packet by the Netfilter layer. As such, the application has six possible operations at its disposal:

- **NF_DROP**: Discards the packet.
- **NF_ACCEPT**: Accepts packet, continuing its path.
- **NF_STOLEN**: The program assumes the packet.
- **NF_QUEUE**: The packet is inserted into another queue.
- **NF_REPEAT**: Reinserts the packet back into the queue.
- **NF_STOP**: Accepts the packet, but ignores existing rules.

These flags provide better flexibility to change the network behavior, thus providing the means for traffic shaping by giving the user the ability to implement its own traffic shaping.

Similarly to TC, Netfilter also suffers from the same limitation, scalability. With the addition of new rules, its processing overhead stops being negligible [57], requiring each packet to execute every rule to find a valid match.

2.4.5 Software Defined Networking

In small networks composed by a reduced number of network devices, the manual configuration of the network can be relatively simple. On the other hand, the addition of new devices increases network complexity, leading to more fine-grained network configuration which cannot be easily handled without significant risk of network configuration error. To mitigate this problem and allow the management of large and complex networks, an administrator needs to resort to automated systems such as Software Defined Networking (SDN) [58].

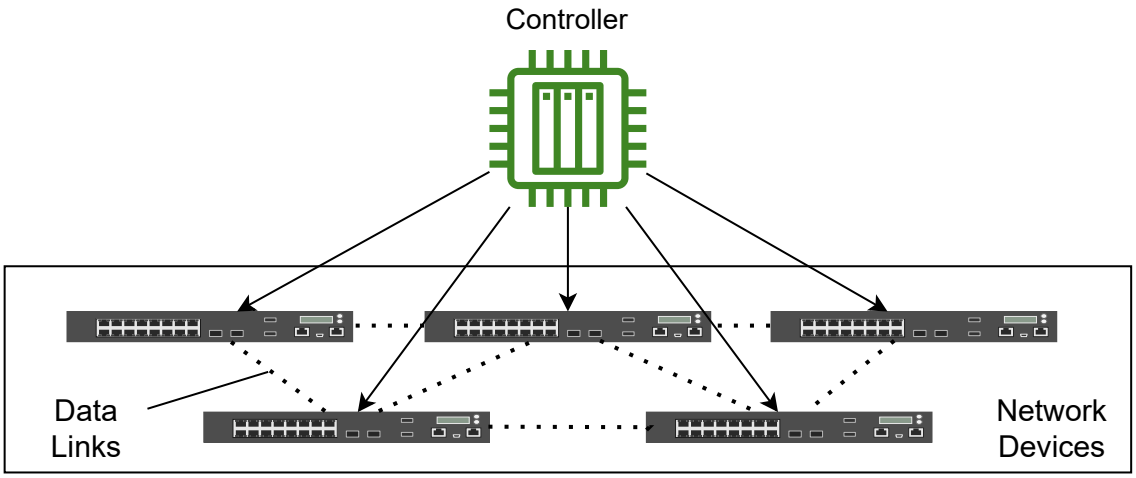


Figure 2.7: Typical software defined network architecture

Software defined Networking emerged as a solution to network engineers to configure complex networks and not be constrained by proprietary software by device manufacturers, as described by Tanenbaum et al. [58] and Kreutsz et al. [59]. By resorting to SDNs, there is a separation of network operations into two layers: the control and data planes. The control plane manages the network configuration and traffic forwarding mechanisms, while the data plane is responsible for applying network rules to manage the traffic flow. Figure 2.7 presents the typical configuration of an SDN, where there is an SDN controller present in the network, responsible for configuring existing network devices.

The separation of the network in these two layers results in the ability to centralize the configuration of the network to the controllers, which are aware of the entire network topology. This knowledge permits the management of forwarding, filtering and prioritization of rules and behavior for traffic while also being aware of the optimal paths [60] to specific network devices.

The controller manages the network, being responsible for sending the correct traffic

rules for the present network devices. When a network device encounters a packet with an unknown destination in the flow table, they request information to the controller. The controller replies with the relevant action to execute, populating the network device flow table.

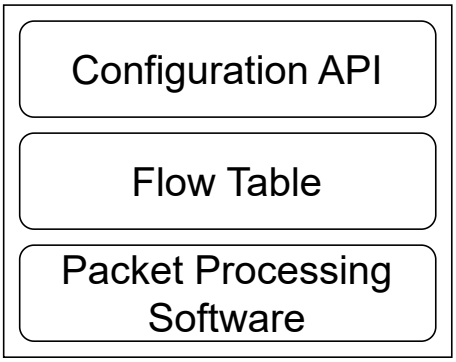


Figure 2.8: SDN device architecture

Figure 2.7 presents the typical SDN architecture of a network device, containing three layers, the configuration API, the flow table, and the packet processing software. The Configuration API is utilized to configure the network device with the correct network configurations pertaining that device, configuring the flow table with the correct routing rules for the observed traffic. With the flow table configured, the network device will apply those rules with the configured packet processing software. The protocol defining the communication of the controller and the network devices, known as OpenFlow [61], enables the configuration of the network between controllers and network devices.

Network emulators such as Mininet [12] and Maxinet [29] leverage SDNs to emulate a virtualized network environment, through Open vSwitch [62], a virtualized network switch that implements the OpenFlow protocol. These tools demonstrate the practical applications of SDNs in creating scalable and flexible testing frameworks, particularly in scenarios requiring dynamic reconfiguration and testing large-scale network environments.

2.4.6 Virtualization and Containerization

As previously explained in Section 2.2, deploying the necessary infrastructure to test large networks is an arduous process. Adding to this already difficult task, another problem for developers is the deployment of a distributed application across heterogeneous machines, ensuring their correct execution. An alternative to deploying an application to bare-metal, is to resort to Virtualization. Virtualization offers an efficient approach deploying multiple network components into a single machine, thus utilizing system resources more effectively.

A network component, in a virtualized approach, is executed inside a virtual machine (VM). Virtual machines are their own operating systems, executing their own filesytems,

RAM, and CPU, isolated from other virtual machines. These virtual machines are referred as Guest OSes, while the software responsible for the management of these Guest OSes are known as hypervisors.

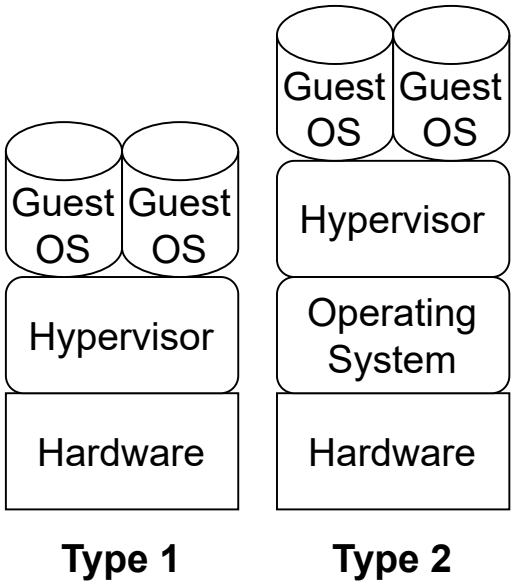


Figure 2.9: Types of a Virtual Machine Manager's architecture

Figure 2.9 illustrates the distinction between two types of hypervisors:

- **Type 1 Hypervisors** execute directly on top of hardware, offering better access to hardware resources and improved performance.
- **Type 2 Hypervisors** runs on top of an existing operating system, adding an extra logical layer, resulting in lower performance when compared to type 1 Hypervisors.

Type 1 Hypervisors typically require support from the underlying hardware, while Type 2 hypervisors are compatible with commodity hardware since they are dependent on the host operating system. This difference in performance between hypervisors is due to the translation of system operations between the GuestOS and the host, where Type 1 hypervisors can provide the GuestOS direct access to hardware in an isolated manner, while type 2 hypervisors require the translation of an operation to the host operating system [63].

In case of deploying a distributed application, which does not rely on the underlying hardware, one can go a step further and deploy an application in a containerized manner. Containerization operates similarly to a Type 2 hypervisor, leveraging the host operating system resources by deploying containers instead of virtual machines. Unlike virtual machines, where the isolation occurs at the operating system level, the isolation is done at the application level, thus using the host operating system resources, improving application performance.

Tools such as Docker [27] or Linux LXC [64] use cgroups [65] and Kernel namespaces [66] to achieve isolation of a process, limiting the visibility of other processes executing in the system while providing their own environment, including independent network configuration and filesystems.

Both approaches provide mechanisms capable of creating a virtualized network without requiring physical implementation. Each approach has its advantages and disadvantages, with the most notorious being performance. As demonstrated in [67–69], containerized solutions have lower resource usage compared to Virtual Machines. For example, an application like the NoSQL database Cassandra [70] demonstrate higher transaction throughput in a containerized environment than in a virtualized one [71].

2.4.7 Scalability

Section 2.4.6 details how leveraging virtualization or containerization technologies abstracts network devices into a single machine. However, for large network topologies, the hardware limitations of a single machine become apparent, limiting testing capabilities. Overcoming these restrictions requires scalable solutions, enabling the distribution of the workload across multiple machines while ensuring correct operation.

Containerization tools such as Docker [27] and Linux LXC [64] provide lightweight environments to execute application code without requiring modifications to the underlying system. However, the developer is still responsible for configuring the network to allow communication between containers, whether the communication is done locally or between two machines. To simplify the configuration of host machines, container orchestration tools such as Kubernetes [72] or Docker [73] are employed. These orchestration tools enable seamless communication between containers running in different machines, allowing the distribution of the workload.

While these tools primarily focus on application management, availability through replication and load balancing [74, 75], these features are not relevant for network emulation, and may introduce unnecessary overhead. Therefore, the challenge lies in leveraging the networking capabilities of these orchestration tools while minimizing extraneous resource usage to maintain an efficient and scalable solution.

2.4.8 Discussion

In the previous sections, several tools and technologies capable of manipulating network traffic. Tools such as XDP and eBPF enable high-performance packet processing before packets are handled by the Linux kernel, while TC and Netfilter allow packet classification, filtering and traffic shaping based on their destination. These fools serve as building blocks in the development of the proposed solution, as they provide mechanisms to configure an emulated network topology with the desired properties. However, each tool has inherent limitations. For instance eBPF is limited in a subset of functions, and when leveraging XDP for traffic redirection, forces the developer to manage the programs

own network stack. Similarly, TC and Netfilter face scalability challenges when used to emulate the properties of network components in a large network, since for each packet it requires the execution of each rule until finding the correct match, introducing overhead.

On a higher layer of network abstraction, Software Defined Networking offers promising capabilities for network emulation. SDNs facilitate the deployment of virtualized networks, mimicking real-world network without requiring specialized hardware. However, the deployment of virtualized devices that implement the desired network capabilities can lead to scalability issues when emulating large-scale networks.

Other approaches were explored regarding the abstraction of an operating system to enable the execution of unmodified application code in heterogeneous environments. Virtualization allows the aggregation of the execution of multiple network devices onto a single machine, by virtualizing the operating system. While effective in reducing infrastructure costs, there is a reduction in performance due to the necessity of translating system operations of a GuestOS to access hardware in the host. Containerization, on the other hand, isolates the application itself, but resorts to available system resources. This approach provides an efficient sharing of host resources between isolated applications. This approach is particularly useful for providing a lightweight environment to run unmodified application code, and by resorting to container orchestration tools, facilitating the communication and distribution of the workload across multiple machines.

2.5 Network Simulators

As discussed previously in section 2.2, the cost of creating, using or managing testing infrastructure presents significant limitations. To address this challenges, simulation or emulation techniques are resorted to model real world network topologies in a more simplified manner. Resorting to simulation allows the user to build a network model in a simulated environment, having complete control over its inputs, controlling how the model behaves. This section presents previous work for modeling network topologies through simulation, describing OMNeT++ discrete event simulator in Section 2.5.1; Section 2.5.2 describes NS-3, a framework for network simulation, requiring the users to implement their own models; Section 2.5.3 presents PeerSim, a network simulator written in Java for simulating peer-to-peer networks; Section 2.5.4 presents Shadow, a discrete network simulator capable of executing unmodified application code. Finally, there will be a discussion regarding their contributions against the defined objective.

2.5.1 OMNeT++

OMNeT++ [10, 76, 77] is a discrete event simulator that models state changes through events. Follows an object-oriented framework approach, facilitating users to build their own simulation according to their needs. OMNeT++ follows a modular pattern, allowing

users to execute their creations as *simple modules*, written in C++, which can be joined to make *compound modules*.

Modules communicate through messages, where communication occurs inside compound modules or between compound modules. Messages are sent through *gates*, which act as module interfaces and *Links* serve to connect modules, providing control over the network properties. The structure of the simulation model is defined through a declarative language called NED.

OMNeT++ has been applied across various domains, including modeling wired and wireless communication networks [78, 79], network switching algorithms [80], and simulating wireless topologies [81].

2.5.2 NS-3

Similarly to OMNeT++, NS-3 [9, 77] is a network simulator that designs the simulation through code, primarly using the C++ language to create models. While OMNeT++ was developed has a network simulation framework, NS-3 focuses on creating accurate models and providing easier debugging methods.

Due to its discrete event architecture, NS-3 models must be developed to process defined events, advancing the simulation time based on the processed events. Since NS-3 contains a model-based architecture, users can design models and distribute it. This approach allows the reusage of existing models, providing a quicker development cycle for prototyping new systems [21].

NS-3 has been proven to be useful to accurately define models, studying their behavior, and ensuring the implemented model correctly aligns to the desired system. Applications include simulating OpenFlow switches [82], routing protocols in datacenter networks [83], and validating wireless networks [20].

2.5.3 PeerSim

PeerSim [84–86] is a java based scalable network simulator with the purpose of simulating peer-to-peer networks.

It offers two types of simulation engines:

- **Cycle-based simulation**: protocols are executed in a predefined order.
- **Event-based simulation**: simulation progresses according to generated events.

PeerSim generates a random network topology depending on the provided configuration file. PeerSim can simulate large network topologies due to existing engines, since the experiment is executed on simulated times, thus providing the capability of computing correct behavior. To achieve scalability, PeerSim does not require events to be processed within a strict time frame to ensure accurate results, in other words, executes in simulated time.

PeerSim architecture revolves around three main components:

- A simulation engine to manage execution flow.
- A configuration manager for setting up simulations.
- A network management logic to oversee the network operation.

Peers in PeerSim are represented as elements in an array structure, where each element tracks the connections to other peers. This modular design allows experimenters to extend the peer structure with functionalities like load balancing, statistics aggregation, and behavior observability in the network.

PeerSim has been leveraged to validate network protocols such as Hyparview [87] which is a decentralized membership protocol, and Plumtree [88] which is a gossip based protocol.

Due to its java-based approach, experimenters are restricted to building their work in java to be able to integrate it into PeerSim for validation. Since peerSim was built for the purpose of simulating peer-to-peer networks, thus is not suitable for scenarios where the underlying network structure is relevant for testing.

2.5.4 Shadow

A more recent network simulator is Shadow [11, 89]. Shadow is a deterministic discrete event network simulator for testing distributed systems. Unlike previous network simulators, Shadow enables the user to execute unmodified application code, capable of scaling to thousands of nodes. It is capable of executing unmodified application code, integrating in a simulated environment by replacing system API calls with its own functions.

In its simulated environment, Shadow abstracts the network by defining it by its more relevant properties, configuring through a weighted graph. In a graph, *vertices* corresponds to network locations, and *edges* define the network paths between network locations. Furthermore, the integration of an application in this simulated environment occurs by connecting a *host* to one of the *vertices*.

For any given network topology, Shadow enables the configuration of certain network properties, restricted by their respective component. The user can configure the bandwidth for a given *vertice*, affecting individually each *host* connected to it. For *edges*, the configurable network properties are latency, packet loss, and jitter.

Shadow has been leveraged to research Tor Networks [90], and has been used by the Tor Project [91] for testing and validation [92].

However, Shadow presents some limitations. One limitation the user will find when executing code under Shadow is that the simulation will stop if the executing code contains any busy loop. This deadlock in the simulation is due to its event driven approach, which is dependent on the system calls it intercepts. If the program is executing a busy loop,

waiting on some state change which does not require using a system call, then Shadow does not advance the simulation. Shadow provides mechanisms to avoid such scenarios, by incrementing some simulation time for every system call, thus continuing the simulation but in turn can impact simulation results.

Shadow is also limited in the configurability of the network topology. For one, it does not model bandwidth congestion, and does not allow changes to the network topology during execution of a simulation, limiting testing capabilities for distributed systems.

2.5.5 Discussion

As presented throughout the previous sections, the simulators described previously provide the capability of simulating network topologies with the desired properties. Furthermore, the ability to execute an experiment in simulated time allows testing systems in large network topologies, surpassing hardware limits by increasing execution time. OMNeT++ and NS-3 better model network components, while PeerSim and Shadow abstract the network topology to its most basic features.

Although these simulators are appropriate to test large systems, it provides no guarantees whether the system behaves similarly against real world-events, due to every network simulator not accounting for unpredictable behavior. To further exacerbate this issue, the experimenter requires adapting programs onto these simulators, which further complicates development, and provides no assurances the real-world implementation follows the tested behavior. Shadow on the other hand provides the capabilities of executing unmodified application code. However, it is clearly limited on providing mechanisms for the user to interact with the network topology and change its state during an experiment, thus limiting testing scenarios.

To summarize, after researching the various network simulators and investigating whether they can achieve the objectives proposed in Chapter 1, the appropriate approach is leveraging emulation, since the described network simulators presents some drawbacks that cannot be ignored.

2.6 Network Emulators

As discussed previously, network simulators, although capable of simulating large network topologies with (at least some) desired properties, suffer from limitations that are not possible to avoid. Another avenue to solve the challenges of creating a testing environment capable of scaling large network topologies, is to leverage emulation. Network emulators have been created throughout the years, with the objective of modeling certain aspects of the network. Network emulators, as opposed to network simulators, are designed to interact with the real world as if it was a real system, resulting in a more accurate model.

In the following sections, related work regarding network emulators is presented. Section 2.6.1 presents ModelNet, a complex network emulator capable of abstracting the network and integrating unmodified code into the experiment; Section 2.6.2 describes Mininet and its variants, network emulators capable of emulating network topologies by resorting to Software Defined Networking; Section 2.6.3 discusses Kollaps, a decentralized scalable network emulator that uses relevant technologies for network emulation; Section 2.6.4, presents Symphony, a network emulator similar to Kollaps but leverages TC differently to allow for large scale experiments. Finally, a discussion about the mentioned network emulators, evaluating their contributions against the objectives defined.

2.6.1 ModelNet

ModelNet [18] is a network emulator designed to test distributed systems, emulating network topologies and their properties. It was designed with a distributed approach, separating the execution of applications, *edge nodes*, from the emulation, *router cores*. Fig. 2.10 shows ModelNet typical machine layout for emulation. In this configuration, unmodified code is executed on *edge nodes*, unaware of the network configuration, by packaging the application into a *virtual node*, with its own IP address. The network traffic goes into the network router and is distributed to the relevant *router cores*, emulating the network topology, applying traffic shaping, namely bandwidth constraints, latency, packet loss, and queuing disciplines.

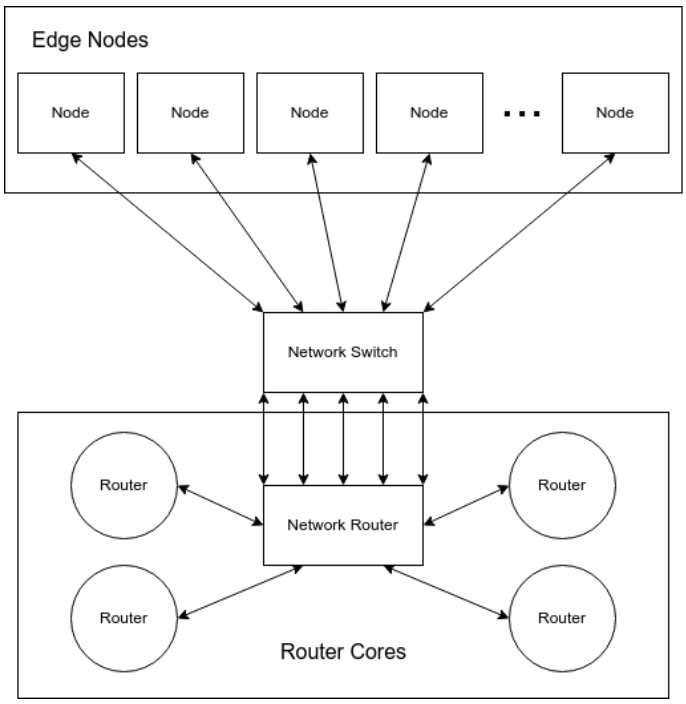


Figure 2.10: ModelNet deployment architecture

To distribute the load between multiple *router cores*, ModelNet uses queues with the desired network properties and queuing discipline, modeling the desired network

topology.

To deploy a distributed system onto an emulated network topology, the experiment goes through five phases, that are as follows:

Creation The user must define the network topology in a GML (graph modeling language) file, with the desired network properties.

Distillation ModelNet processes the provided network topology, configuring the emulation and optimizes the emulated network structures, emulating the network topology, which is distributed amongst the *router cores*, ensuring the minimizing of information passing during emulation.

Assignment ModelNet configures the *router cores*.

Binding To deploy applications, ModelNet encapsulates the application into a *virtual node*, assigning it to an *edge node*. After distribution of *virtual nodes* amongst the *edge nodes*, ModelNet calculates for each *virtual node*, all the shortest path routes to all the other *virtual nodes*, computing a routing matrix that is installed in every *router core*.

Run starts the experiment.

During the experiment, ModelNet emulates the network topology, intercepting the network traffic from a *virtual node* and redirecting to the relevant *router core*. In a *router core*, the traffic shaping is applied according to the path towards the desired destination. Before network traffic reaches the designated *virtual node*, the traffic shaping might occur through multiple *router cores*, where each *router core* emulates part of the network topology.

This approach provides the researcher the ability to test large distributed systems, by distributing the load of the applications and the emulation, allowing scalability. However, ModelNet contains some limitations, for example, its deployment. Due to the separation of responsibilities of application execution and emulation, it is required the presence of a network switch and a network router to connect machines, which not only leads to a physical limitation of network throughput, but also the link sharing of the physical link between all *virtual nodes* present in an *edge node*.

Another limitation of ModelNet is its approach to emulation. As described earlier, ModelNet follows a series of steps to configure the emulation, optimizing its configuration to minimize cross communication between *router cores*. As such, it is not possible to modify the network topology during execution as that would require reconfiguring the emulation again, and recalculating all the paths for each individual virtual node.

2.6.2 Mininet

Mininet [12] enables researchers and experimenters to experiment with Software Defined networks and OpenFlow [61](see Section 2.4.5). Mininet is written in Python and executes the Mininet Runner to manage the emulated network. The users can interact

with this runner through a CLI during execution. Another approach is to define the desired network topology by importing Mininet's Python libraries and configure their own custom experiments.

Differently than other network emulators, Mininet leverages existing Linux mechanisms to configure the emulated network. It resorts to lightweight process isolation with cgroups [65] and namespace [66] creation, reducing emulation overhead, and Linux Virtual Ethernet Pairs [93] to connect network namespaces. Mininet provides three levels of API operations the developer can resort to and are as follows:

- **Low-level API:** Allows the user to configure individual processes, switches and links. Designed to create a network.
- **Mid-level API:** Provides an API to instantiate hosts, switches and links in an abstract manner. Useful for managing a network.
- **High-level API:** Enables the creation of full topological networks, in a parametrized approach.

This approach enables the user to configure the desired network topology, and execute network devices that support OpenFlow. By deploying OpenFlow friendly devices, Mininet provides the researcher the ability to learn, test and evaluate SDN networks, which in turn, can export its configuration to real network devices.

However, Mininet contains some limitations. Due to its design, Mininet is restricted to one machine, and thus cannot scale its execution to other machines. Another restriction of using Mininet is the user is responsible for configuring the network devices on the emulated network, when considering large network topologies, as well as defining the desired network properties for traffic shaping.

Work has been done to address some issues, for example, Maxinet [29] addresses single host limitation by running multiple Mininet instances, connecting them through Generic Routing Encapsulation (GRE) tunnels [94], consisting of encapsulating the network traffic to redirect to other machines. ContainerNet [95], on the other hand extends Mininet by adding Docker to Mininet, allowing the execution of containerized applications.

2.6.3 Kollaps

Kollaps [13, 96] is a decentralized network emulator with the objective to emulate large-scale distributed systems while ensuring desired network properties.

Kollaps enables the description of the network topology in a file, converting the described network to rules to be applied by TC [44] (see Section 2.4.3) during experimentation. To avoid executing routers and switches, Kollaps rearranges the network through *network collapsing*, by calculating the shortest path for every endpoint and aggregating

the network properties of each path, thus reducing computation required while maintaining end-to-end properties. The network rearrangement occurs at the beginning of the experiment.

Kollaps leverages TC to apply end-to-end link properties between hosts such as bandwidth limitations, latency, packet loss and jitter. Since Kollaps simplifies the network at the end hosts, it requires real-time modification of TC rules to emulate link congestion. Such work is done by an *emulation core*. The *emulation core* observes the bandwidth usage, and shares it with other emulation cores to apply a fair share of bandwidth for connections in the same routing path.

With the presence of an *emulation core* in each container running, Kollaps enables dynamic behavior by configuring it in the network topology description. When the experiment is running, the emulation core is responsible for applying the dynamic behavior defined by the practitioner. The TC rules to apply dynamic behavior are also configured during preprocessing of the network topology.

Kollaps leverages technologies such as Docker [27] and Kubernetes [72] to simplify the network model. Docker allows the deployment of unmodified application code through containers, and provides mechanisms to allow communication. Kollaps offers two approaches for load distribution. One option is by resorting to Docker Swarm, or deploying the experiment in a Kubernetes environment. This allows Kollaps to evenly distribute containers through all participating machines, and does not require managing network traffic between machines.

As previously described, Kollaps contains the necessary mechanisms to emulate large network topologies, but by collapsing the network, it restricts the ability to dynamically modify a network during experiment. Changes to the network topology in this scenario results in pre-computing new TC rules and new shortest paths, which could introduce large overhead for a simple change. Furthermore, Kollaps also restricts experiments to only containing no more than 65535 network links, constraining scalability. For example, in fully connect network topologies, this limit restricts the number of containers in an experiment to a maximum of 256.

2.6.4 Symphony

Symphony [14] is a distributed network emulator similar to Kollaps. It follows the same approach as Kollaps and ModelNet, also requiring the user to define the network topology in file. For an experiment, it will then convert that file to the desired network topology. Symphony also resorts to Docker [27] for its containerization technologies and the capability of configuring the network properties for each individual container.

For a given experiment, Symphony follows a series of phases, that goes as follows:

- **Initial:** Processes the provided network topology, registering the desired applications, links and routers.

- **Setup:** Distributes the information amongst the participating machines, and configures Docker and stores the relevant Docker images.

- **Execution:** Symphony signals all participating instances to start the experiment and apply all relevant TC [44] rules to emulate the desired network topologies.

Symphony stands out from Kollaps by providing a better approach in topology definition and optimizations to the TC rules, where Symphony generates rules depending on the communication scenario, which can be server-client or peer-to-peer communication. In server-client communication, Symphony does not emulate the network for communication between clients.

However, by utilizing TC, Symphony also suffers from the same restrictions as Kollaps. By following an experiment approach, it restricts the researcher of interacting with the emulation during execution, since all new behavior must be defined before execution.

2.6.5 Discussion

Through the previous sections, multiple network emulators were presented regarding their objective and used approach. The objective of this work is the proper emulation of large network topologies and provide mechanisms to interact with the emulation to provide new testing capabilities to the user. As such, the presented network emulators provide creative solutions to solve some of the challenges regarding the defined objectives.

ModelNet [18] presents an early approach to emulating large network topologies. By separating the application execution from the emulation, ModelNet can define routing rules to redirect the network traffic to the correct *router core* for network emulation. However, this approach demonstrates that cross traffic between *router cores* leads to performance degradation. Another limitation is the machine configuration, however, this limitation can be mitigated by utilizing existing technologies such as Docker [27] or Kubernetes [72].

Mininet [12] has their focus in emulating network topologies through Software defined networking capable devices, providing an environment for researcher to test and learn about data-center networking. Consequently, for testing distributed systems in a Mininet environment requires the user to configure the network devices emulated. However, this constraint demonstrates a solution to emulate network topologies by simplifying the network devices to its most basic functions, and employ a routine similar to OpenFlow to fetch and configure routing rules.

Kollaps [13] and Symphony [14] provide a great concept to network emulation. By utilizing Docker and Linux Traffic Control, they are capable of emulating large network topologies, demonstrating the viability of such tools that can be leverage in this work. However, their experiment-based approach limits the capabilities of introducing dynamic behavior to the network, requiring the behavior definition at the beginning of the experiment.

To summarize, all the presented tools provided great inspiration for the design and implementation of a new network emulator capable of emulating large network topologies to test and evaluate distributed systems, while providing mechanisms to dynamically modify the network.

2.7 Summary

Throughout this chapter, subjects related to networking, its concepts, tools, and related work have been presented, providing context and inspiration for the work done. The chapter starts by explaining simple network concepts, starting from the communication between two machines, to the constitution of large network topologies, requiring various types of network mechanisms, that can be described by the OSI Model. Next, the chapter describes possible approaches to test large distributed systems, discussing the usage of research testbeds or resorting to the Cloud, concluding that these tools are not appropriate for testing since due to lack of availability (research testbeds) or high monetary costs (Cloud). This is turn, leads to resorting to techniques such as Simulation or Emulation. These techniques allow for accurate testing in a smaller scale, without requiring large and complex infrastructure.

To complete these techniques, existing tools were explored, enabling the developer to modify network properties and apply traffic shaping. The chapter glosses over XDP and eBPF, which when used in tandem, enables the user to have complete control over the receiving network traffic, at the cost of requiring the re-implementation of existing mechanisms of the network stack; while Linux Traffic Control (TC) and Netfilter provides the user the ability for traffic shaping and traffic manipulation.

For traffic routing, Software Defined Networking (SDNs) was presented, providing a solution to implement routing regardless of the underlying network device.

With network topologies covered, the chapter discusses solutions to execute unmodified application code, resorting to virtual machines or containerization, where it was concluded that containerization provides a better lightweight environment to execute unmodified applications while providing isolation mechanism and configuration.

Finally, the chapter ends by presenting related work regarding testing distributed systems, starting with network simulators and their design, and due to its constraints, conclude the choice of resorting to emulation instead of simulation.

Following network simulators, network emulators were presented, describing their approach to emulating large network topologies, their limitations, and their approaches inspires the development of this work.

In conclusion, the tools discussed and the various approaches taken from existing network emulators, provide us with a strong foundation on how to tackle some problems and challenges that may arise during the development of the network emulator.

Given the limitations of the various solutions presented throughout this chapter, we choose the emulation route, as it provides the most fidelity between testing and

real deployments. Furthermore, we tackle the limitations of the current state-of-the-art network emulators–the lack of control for dynamic changes of the network topology and properties at runtime. In the following chapter, we present our contribution in detail.

3

GO Network Emulator

In this chapter we introduce GONE, a scalable network emulator written in Golang that provides an accurate and dynamic network topology, enabling the user to test and evaluate distributed systems. To better recreate the conditions of the deployment of a system in real-world conditions, GONE provides the means to execute unmodified application code by leveraging containerization technologies such as Docker, and uses eBPF and XDP for traffic interception, enabling the emulation of the desired network topology.

We first start this chapter by revisiting the established objectives for this work in Section 3.1; Section 3.2 presents GONE and its subsystems, providing a high-level description of each individual component; Section 3.3 presents the existing API the user can leverage to operate the network emulator. Next, in Section 3.4, we detail the internal processes that allow for network emulation. Finally, we discuss the limitations in our work in Section 3.5.

3.1 Requirements

As mentioned in Chapter 1, distributed systems are increasing in complexity, requiring the creation of inter-cooperation and coordination mechanisms to ensure correct execution. This increase requires the existence of testing tools that allows developers to correctly validate their solutions.

Testing a single application is a simple task when testing for correctness. The problem arises when testing a distributed system where an application requires communicating with multiple instances, where the network conditions can affect system functionality.

As an initial approach, practitioners can resort to existing testbeds that offer realistic deployment scenarios, or create their own testbed by leveraging Cloud solutions. However, resorting to an existing testbed often limits testability due to limited infrastructure, typically shared amongst other users. Creating a dedicated testbed in the cloud on the other hand, constrains the user to the configurability provided by the services used and can lead high monetary costs.

Another approach researchers can leverage is simulation or emulation. By resorting to simulation tools, the user can create a simulated environment containing large network

topologies. The main problem with simulation is that it requires reimplementing the application for the simulator, which is both an additional effort, but also allows for human error. Furthermore, the simulator might not accurately replicate the real environment, a problem known as the *reality gap*, which can lead to a discrepancy between testing and real deployment.

We choose emulation, as it allows for a more realistic approach and provides more accurate results, since a distributed application is not aware it is interacting with an emulated environment–the emulator runs real code. By simplifying the network model, only emulating its most relevant functionalities while retaining accuracy, it is possible to emulate large network topologies before reaching hardware limits. To further mitigate reaching said limits, it is necessary to design the emulation to be scalable, distributing the load across multiple machines, while also providing a dynamic environment the users can interact with to make any desired changes to the topology. For that allowing to emulate certain aspects while also interacting with the real world environment. Thus, we present GONE, a scalable network emulator written in Golang that provides a dynamic and accurate networking environment where users can build their desired network topology and test their systems.

Specifically, GONE was designed to fulfill the following requirements:

- **Network Abstraction Layer:** Design a network abstraction layer capable of emulating desired network properties regarding bandwidth limits, latency injection, jitter, packet loss, and link congestion, ensuring accurate behavior.
- **Dynamic Reconfiguration:** Allows real-time configuration of network topology and link properties during runtime with minimal disruption of the natural flow of execution.
- **Scalable:** Design approach to allow the emulation of large-scale topologies by providing mechanisms for distributing the load across multiple machines.
- **Support for Real Application Code:** Execute arbitrary, real, and unmodified application code by leveraging containerization technologies.
- **Ease-of-use:** Require minimal dependencies to deploy the network emulator, or its interaction.
- **Extensible:** Provide mechanisms to allow the user to create more complex network operations.

3.2 Architecture

The system has 3 major components. It is divided into GONE [97], GONE-Proxy [98], and GONE-RTT [99]. These components are written in Golang, and leverages XDP sockets

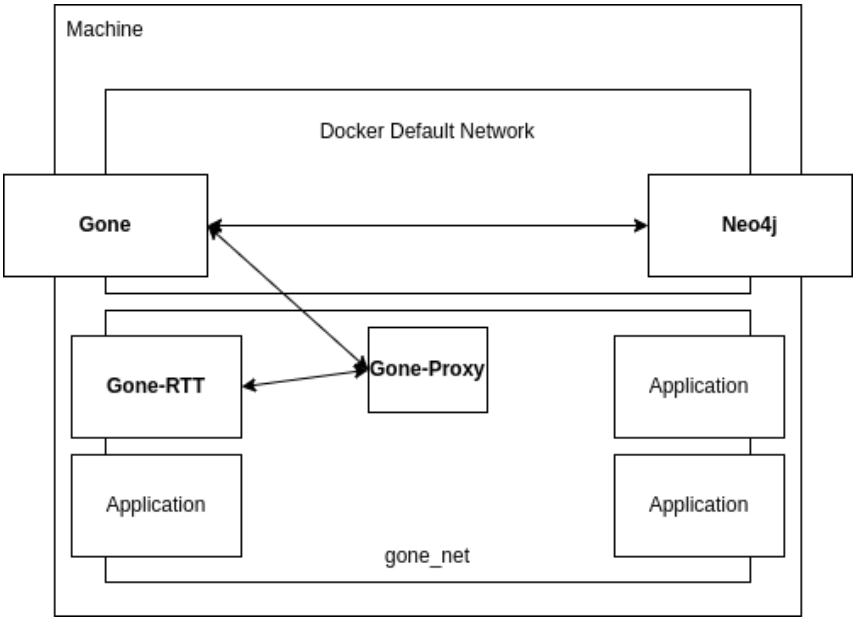


Figure 3.1: Gone Deployment in a containerized environment

(see Section 2.4.2) and eBPF programs (see Section 2.4.1) for packet interception, Docker for its containerization technologies, and neo4j [100], a graph database, to manage the network components of the emulation. Figure 3.1 demonstrates the typical deployment of the GONE system in a containerized environment in a single machine and the communication patterns between the existing components.

GONE-RTT, containing 81 SLOC, and is responsible for bootstrapping the docker network and be an endpoint from GONE-Proxy to measure the delay of the Round-Trip Time between a containerized application and GONE-Proxy in order to optimize the delay applied in the emulation to account for delay GONE cannot control after the network traffic leaves the emulation.

GONE-Proxy, written in 1003 SLOC, intercepts the network traffic between containers and sends it to the GONE application. It is responsible for intercepting the traffic between containers in an individual machine by leveraging eBPF programs and XDP sockets in order to redirect the network traffic to GONE. It also propagates information regarding the observed delay calculated between itself and GONE-RTT to GONE.

GONE, comprises 11281 SLOC. It is responsible for the management of the emulated network, managing the emulation across different physical machines and interacting with docker to launch the desired applications. It can be interacted with through the use of its available API.

Auxiliary tools were created to complement GONE, GONE-CLI [101] and GONE-Agent [102]. GONE-CLI is a Command-Line Interface created to simplify the interaction with the emulator by converting user commands into the correct HTTP requests in order to configure the emulated network topology. GONE-Agent is a simple agent that can be executed alongside GONE, when deploying GONE in a cluster of machines. This agent

allows the user from a single machine to restart or shutdown all participating machines, simplifying management.

3.3 GONE API

In this section, we detail the available API to interact with the emulator. Annex II.1 shows all the available API endpoints provided by GONE.

We start by detailing the operations to add network components to the emulation, consisting of adding nodes, which represent the addition of an application to the emulation, and the addition of bridges and routers for network topology creation (Section 3.3.1). Next, we present the endpoints to allow the linkage between network components (Section 3.3.2).

To provide a more dynamic emulation, we detail network operations the user can resort to temporarily disrupt a particular network component (Section 3.3.3).

To provide mechanisms to extend the capability of introducing new network behavior, we detail the endpoints that allow a user to externally receive network traffic, allowing the creation of custom behavior of a network link (Section 3.3.4).

Lastly, we describe information endpoints, allowing the user to obtain information regarding a particular network component and obtain information about its connections (Section 3.3.5).

3.3.1 Network Components

To add a new component to the emulation, the user can resort to the available REST endpoints shown in Listing 3.1, and its removal counterparts.

Listing 3.1: Add and Remove Operations

```
"/addNode"
"/addBridge"
"/addRouter"

"/removeNode"
"/removeBridge"
"/removeRouter"
```

GONE allows the creation of network nodes, network bridges, and network routers in the emulated network model. The network node allows the user to add a new container to the emulation, by providing the relevant docker command to execute. To promote locality, the user can add a network bridge to the emulation, and utilize this structure to aggregate a set of network nodes, allowing local communication without requiring the usage of a network router. Lastly, there is also the addition of a network router, allowing the creation of more complex network topologies, configuring routing rules to allow communication between network nodes in the network.

In a distributed setting, the user can supply a machine identifier, representing another GONE instance, to remotely create the desired network component.

3.3.2 Connections

Another set of operations that are available to the user is the configuration of links between network components. Listing 3.2 shows the available endpoints for connecting and disconnecting network components.

Listing 3.2: Available operations for connecting or disconnecting network components

```
"/connectNodeToBridge"
"/connectBridgeToRouter"
"/connectRouterToRouter"

"/disconnectNode"
"/disconnectBridge"
"/disconnectRouters"
```

The user can connect a node to a bridge, a bridge to a router, and connect two routers. In a connect operation, the user can define the network properties of the newly created link, being able to configure latency, bandwidth, drop rate, jitter, and link weight.

Changes to the links in the network are not immediately observed throughout the entire emulated network, and can be forced to be visible by propagating the routing rules of a given router (see Section 3.3.3).

3.3.3 Network Operations

To provide a more dynamic network environment, GONE provides operations to temporarily stop a connection from working, emulating a loss of connection to a given component. Listing 3.3 shows the available operations to disrupt a network link between two components.

Listing 3.3: Available operations for disrupting network components from the emulation

```
"/disruptNode"
"/stopDisruptNode"

"/disruptBridge"
"/stopDisruptBridge"

"/disruptRouters"
"/stopDisruptRouters"
```

The user can temporarily disable the connection between a node and bridge, the connection between a bridge and a router, and the connection between routers.

Listing 3.4: Available operations for disrupting bridges and routers

```
"/startBridge"
"/stopBridge"

"/startRouter"
"/stopRouter"
```

```
"/forget"
"/propagate"
```

Besides disrupting connections, the user can also disrupt bridges and routers, as shown in Listing 3.4. The user can temporarily stop the desired bridge or router in the emulation.

Besides disruption, the user also has available two other commands, */forget* and */propagate*. The user can resort to the */forget* operation to drop the routing rules of a given network router, in order to calculate and configure new routing rules to better reflect the network topology state. */propagate* is another operation the user can leverage to spread the routing rules of a given router throughout the network. Leveraging this operation before any experiment avoids configuring routing rules in routers during the experiment, leading to higher observed latency than expected.

3.3.4 External Network Operations

To provide more flexibility in testing and validating distributed systems, the network emulator provides two more operations the user can leverage to implement custom network behavior, externalizing the network traffic across a particular link in the emulated network. The user can receive a copy of the network traffic (sniffing) or can directly receive network traffic (intercepting), interrupting the natural traffic flow. Listing 3.5 shows the available endpoints.

Listing 3.5: Available operations for sniffing or intercepting network traffic

```
"/sniffNode"
"/sniffBridge"
"/sniffRouters"
"/stopSniff"
"/listSniffers"

"/interceptNode"
"/interceptBridge"
"/interceptRouter"
"/stopIntercept"
"/listIntercepts"
```

The user can *sniff* or *intercept* network traffic between a node and a bridge, a bridge and a router, or between routers. GONE also provides endpoints to the user to check which links are currently being utilized.

Using one of the operations leads GONE to send network traffic to a newly created socket, which the user can create a custom program that connects to it and receive network traffic in order to apply some sort of operation or obtain metrics. In Section 4.4 we present some examples of using these operations to implement new network operations.

In a distributed setting, to receive network traffic from the emulation, the user must execute the custom program in the same machine where the link has been created.

Annex II.2 shows an example of a custom go program that receives as argument the path of the socket, and upon receiving network traffic, prints to standard output the network packets into the PCAP format, and can be read by tcpdump [32], a known tool for monitoring and reproducing network traffic.

Another example is present in Annex II for the intercepting operation. The program opens a connection to the desired socket, and depending on the observed network traffic, introduces a delay of 10 milliseconds to the network traffic with the source IP address of 10.1.0.101.

3.3.5 Network Topology Information

To allow the user to know which connections have been made, the network emulator provides 3 endpoints for querying information, one for each component. Listing 3.6 shows the relevant endpoints for obtaining information of a node, bridge, or router.

Listing 3.6: Available operations for inspecting network components from the emulation

```
"/inspectNode"
"/inspectBridge"
"/inspectRouter"
```

Querying information from a particular node yields information about its MAC address, the bridge it is connected to and the network properties of the link. Inspecting a bridge yields information about the connected nodes and the connect router, detailing the network properties of each link. For a particular router, the user obtains information about the connected routers, and information about all connected bridges and their connected nodes. This information is useful for assessing if the network components are working as expected, or inspecting their current configuration.

3.3.6 Using GONE

In this subsection, we provide an example of how to leverage GONE-CLI to configure our network emulator to deploy an experiment. We use as an example the configuration of the experiment done in Section 4.2.3, where we evaluate the effect of link congestion on an application.

Listing 3.7: Network configuration of two clients and two iperf3 servers

```
gone-cli node -- docker run --rm -d --network gone_net --name server1 nicolaka/netshoot
    iperf3 -s
gone-cli node -- docker run --rm -d --network gone_net --name server2 nicolaka/netshoot
    iperf3 -s
gone-cli node -- docker run --rm -d --network gone_net --name client1 nicolaka/netshoot
    iperf3 -c server1
gone-cli node -- docker run --rm -d --network gone_net --name client2 nicolaka/netshoot
    iperf3 -c server2
```

```
gone-cli bridge cbridge
gone-cli bridge sbridge
gone-cli router r-1
gone-cli router r-2

gone-cli connect -l 5 -w 1G -n client1 cbridge
gone-cli connect -l 5 -w 1G -n client2 cbridge

gone-cli connect -w 1G -n server1 sbridge
gone-cli connect -w 1G -n server2 sbridge

gone-cli connect -w 100M -b cbridge r-1

gone-cli connect -w 100M -b sbridge r-2

gone-cli connect -w 100M -r r-1 r-2

gone-cli unpause server1
gone-cli unpause server2
gone-cli unpause -a
```

Listing 3.7 shows the necessary commands to configure an experiment contain two clients and two Iperf3 servers. This example demonstrates a simple network configuration that will execute two iperf3 clients against the servers. Fig. 4.3 represents the configured network topology.

The first four commands relate to adding a new application into our network emulator. Our network emulator accepts a docker command to be executed on the emulation. The docker command must be provided after the – of the *gone-cli node* command, and must include the *-d* and *–network gone_net* flags in order to successfully launch the container. When adding a new container to the network, the container starts in a paused state, awaiting user input to start the container execution.

Next, there are the operations to add bridges and routers. For these commands, the user only needs to provide a unique id that has not already been used to be able to add a new bridge or router to the network.

To allow communication between applications, it is necessary to connect the components together. By using the *gone-cli connect* the user connects two different components by providing the correct flag, *-n* for connecting a node to a bridge, *-b* for connecting a bridge to a router, and *-r* to connect routers. In this command it is where the network properties are configured. The *-l* flag allows configuring latency, by providing a value in milliseconds, and configuring available bandwidth, with the *-w* flag. The user can provide values like 1G, 100M, 50K to configure 1 Gigabit per second, 100 Megabits per second, or 50 Kilobits per second, respectively.

To start the experiment, it is necessary to unpause the applications in order for them to start. In this example, we first start by unpausing the servers, followed by everything else, by providing the *-a* flag to the *unpause* command.

3.4 Implementation

In this section, we detail the internal logic of each component required to materialize the logic of the operations described previously. We start this section by describing how the network emulation is executed, explaining the different components. Next, we detail the process to add a new container to the emulation, describing the steps and synchronization between systems. Furthermore, we detail how the different instances organize between them, using a Leader-Follower approach, describing their responsibilities.

For managing network topologies and allow traffic to flow from one application to another, we detail how the emulated routers configure their routing tables to connect the various applications. Next, we detail the approach to intercept network traffic from the containers and introduce it into the emulation. To conclude the explanation of our network emulator, we explain the process of obtaining the minimum delay observed between an application and the proxy to better reflect the latency of the network traffic. To finish this section, we describe some limitations encountered or introduced by design choices when developing our network emulator.

3.4.1 Network Emulation

The network emulator was built in Go to leverage its concurrency primitives and message passing mechanisms, since most of the processing occurs in applying traffic shaping and transversing the emulated topology.

By resorting to *goroutines*, GONE parallelizes most of the operations in the emulation. *Goroutines* are software threads managed by the Go runtime [103]. Go manages these threads by distributing it in an M:N scheduler, where M is the number of *goroutines* and N is the number of hardware threads available. The go scheduler is responsible for managing which *goroutines* are executed. Due to this abstraction, the Go runtime has more context regarding the execution of a particular *goroutine*, efficiently deciding which *goroutines* can be executed and which ones are waiting for input. For the network emulator, this functionality is particularly useful since it prioritizes the emulation components that need to process network traffic.

Another internal mechanism the network emulator leverages is *go channels*. This primitive provides the possibility of data passing and synchronization between *goroutines*, due to its internal structure containing a *ringbuffer* to store data and a *mutex lock* for synchronization. The usage of this primitive provides the Go runtime more information about a *goroutine* and its execution, enabling a better routine scheduling.

For the emulation, utilizing *goroutines* and *go channels* is essential to create the emulation data structures, since *go channels* allow the synchronization between network components, as well passing information in an orderly manner.

Fig. 3.2 presents the modelling of the network components. All network components built in GONE follow the same pattern, where each network component contains a *go*

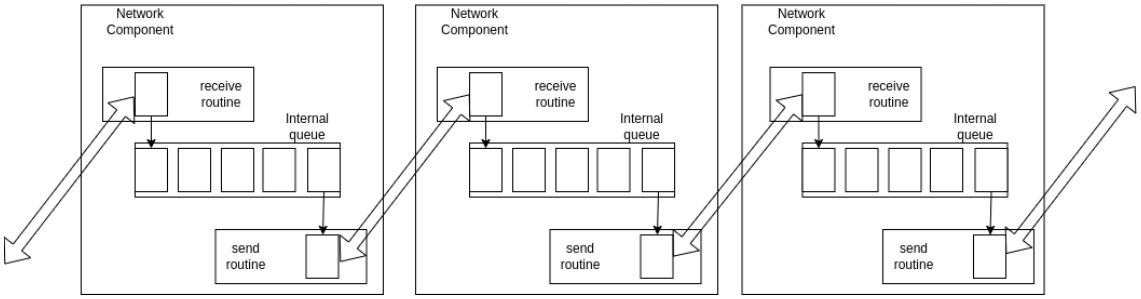


Figure 3.2: Network traffic flow between network components in GONE

channel that receives network packets, an internal queue, and a destination channel, flowing the network traffic through the network emulation.

By leveraging these mechanisms, GONE is able to model nodes, bridges, and routers.

Nodes are responsible for introducing network traffic into the emulation, and managing the connection to the socket created by GONE-Proxy. They also store information regarding which bridge they are connected to.

Bridges are responsible for locally connected nodes. Bridges contain a small routing table with rules to redirect network traffic to any connected node, by associating the MAC address to the relevant node. If the destination address is a broadcasting address, then the bridge broadcasts the network packet to all connected nodes. If it is an unknown address, then the bridge redirects the network packet to the connected router, if it is connected to one, dropping the network traffic otherwise.

Routers, to allow communication between containers in different routers, require more mechanisms. Routers also contain a routing table, but in this scenario matching MAC addresses to the correct bridge. When faced with an unknown destination, routers request the emulation for information about the address, calculating the shortest path (see Section 3.4.4 for details) and updating the routing tables of every affected router with the correct routing rule.

For traffic to flow from one node to another, it is necessary to make connections between components. The links are responsible for applying the desired network properties of a connection. The following section provides more details about the inner workings of the network link.

3.4.1.1 Link

To pass network traffic along network components, the network emulator creates links between components. For each connection between two components, GONE creates two links that represent each direction of the link. It is through the link model that GONE is able to, seamlessly, introduce new behavior that is transparent to the network emulation.

Fig. 3.3 shows how the network traffic is passed along two network components, namely two routers. The traffic that goes from *transmit* channel belonging to one router goes to the network link's Shaper (Section 3.4.1.2), a component that applies the network

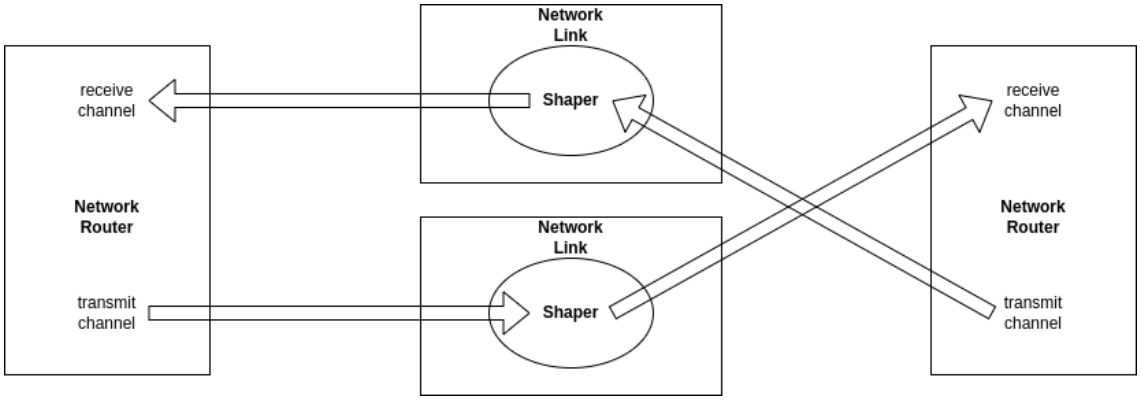


Figure 3.3: Traffic exchange between two components through a Bilink

properties of that link. The shaper also follows similar logic present in Fig. 3.2, where a receiving routine pulls the network packets from the *transmit* channel and stores it in an internal queue. Next, the *send* routine fetches a packet from the internal queue, applies the network properties configured before passing along the next go channel, namely the *receive* go channel of the next network router.

3.4.1.2 Shaper

As previously mentioned, each link contains a shaper, responsible for applying the desired network properties, such as latency, bandwidth, jitter, and packet loss. A benefit of this approach is the possibility to effortlessly modify the shaper present in a network link to perform other operations. GONE implements multiple shapers to provide extra functionality.

- **Null:** Drops all received network packets.
- **Default:** Applies the network properties configured by the developer.
- **Remote:** Redirects network traffic to another emulation instance.
- **Sniff:** Copies the received network traffic and sends it to a local socket.
- **Intercept:** Intercepts the network traffic, redirecting it to a local socket.

The network emulation changes the network shaper depending on the operation the user chooses. For nodes that are not connected to any bridge, the node is connected to a link with the *null shaper*, dropping all network packets, emulating a disconnected network.

Connecting two components on the same emulation instance configures the *default shaper*, applying the configured network properties (see Section 3.4.1.3). With multiple GONE instances, the connection between two routers in different machines will instead be the *remote shaper*, redirecting the network traffic to the destination instance (see Section 3.4.3.2).

To utilize the *sniff* and *intercept* shapers, it is necessary that exists a link between two components utilizing the *default shaper*. GONE upgrades the *default shaper*, and copies or redirects network traffic to a socket, for an external program to receive it.

3.4.1.3 Network properties

Since GONE does not resort to TC to apply network properties, GONE intercepts the network traffic at the lowest level of the Linux network stack and thus is responsible for its management, requiring the full implementation of the relevant network properties. Gone implements five network properties, namely latency, bandwidth, jitter, drop rate and link congestion.

GONE implements each network property as follows:

- **Latency:** A simple sleep function with a defined latency value.
- **Bandwidth:** A token bucket algorithm to control the transmission rate of the network link, providing the capability of limiting the throughput of a link when multiple traffic flows pass through the same link.
- **Jitter:** Simple uniform distribution with the mean and standard deviation configured with the values of latency and jitter respectively.
- **Drop Rate:** Generates a pseudo-random float value between 0 and 1 and checks against the configured value.
- **Link Congestion:** Network traffic from multiple sources are put in the same network link.

3.4.2 Bootstrapping an Application

To integrate an application in the emulation, GONE and the proxy need to synchronize in order to successfully bootstrap an application.

Fig. 3.4 demonstrates the steps and messages executed between Gone and Gone-proxy to add a new container to the emulation.

To add an application to the emulation, GONE executes the provided docker command and immediately pauses the container, stopping its execution and allowing the configuration of the container network interfaces in the network namespace. When proxy executes inside this network namespace, it has no information about the IP or MAC addresses of a given container to be able to send network traffic to the correct interface. To circumvent this issue, the proxy runs a HTTP server which the network emulator can use to signal the addition of a new container to the emulation. Upon the signal, it discovers the new network interface and configures eBPF and XDP to intercept traffic. However, in this situation, the proxy does not know who this network interface belongs to, and thus cannot send network traffic to this interface.

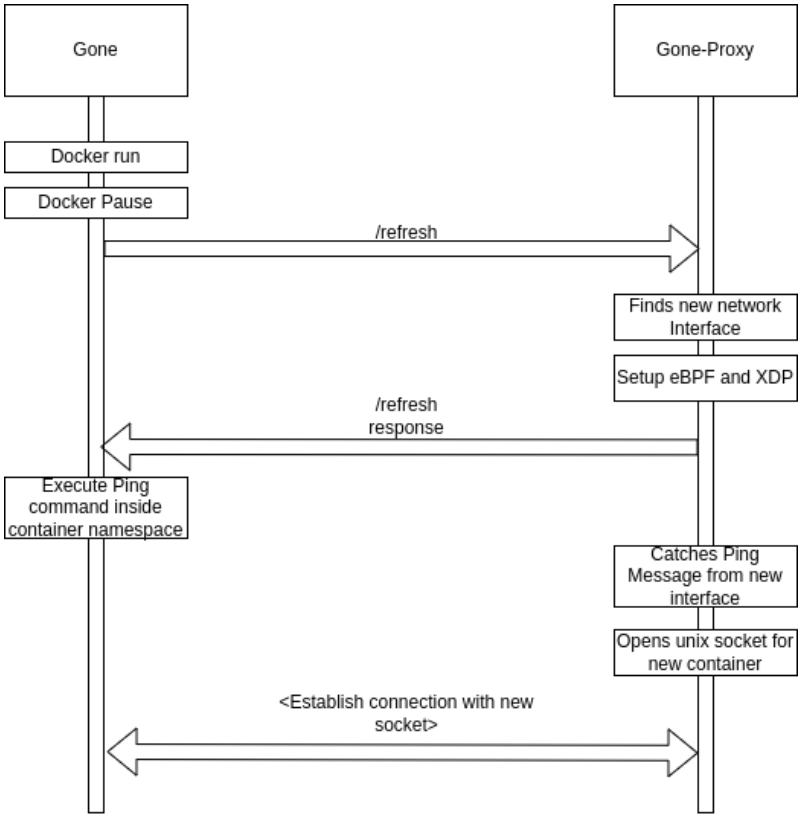


Figure 3.4: Routine to Bootstrap a container in the emulation

For the proxy to associate a given network interface to a container, GONE executes a ping command inside the network namespace of the container. The proxy sees the ping traffic, and extract from those packets the relevant MAC address. With this information, it configures a new socket with the found MAC address as an identifier, where it sends the network traffic to. GONE also makes some changes to the network interface of the container, disabling the offloading of the TCP checksums to hardware (see Section 3.5 for details) and updates the ARP tables with information of all the containers present in the emulation.

After the necessary configurations, GONE updates the ARP tables of the containers with the new information.

If executing the emulation in a distributed fashion, GONE sends a message to the other instances to update their containers ARP tables with information regarding the new container.

3.4.3 Distributed GONE

To provide scalability to the emulation, allowing the emulation of large network topologies, GONE contains mechanisms to distribute the network topology across multiple instances. GONE follows a Leader-Follower communication pattern, where the leader broadcasts the operations to other instances.

The responsibilities of the leader are described as follows:

- Contact node for a new instance to contact and receive information about other instances to establish connections.
- Distribute user operations to the relevant instance.
- Broadcast information to other instances.

The followers on the other hand, have fewer responsibilities:

- Await to receive operations from the leader.
- Responsible for the management of the components created in its instance.

However, any network emulator instance is responsible for managing their own part of the network topology assigned to it. The leader contains extra information about the created components and where these components are located, but has no information regarding the connections made between components in other instances.

3.4.3.1 Connecting Instances

When a new GONE instance tries to join the emulation, it establishes connections to all other instances. To further reduce the observed latency due to processing overhead of the emulation, every instance sends a configurable number of requests to other instances and measures the latency of the operation. The minimum value of the latency observed is used to offset latency injection in the emulation, since redirecting network traffic to another network emulator introduces a varying delay.

3.4.3.2 Remote Traffic Management

To allow for traffic redirection to other GONE applications, it is necessary to have mechanism that allow that redirection. As mentioned in Section 3.4.1.2, GONE implements a *remote shaper* that allows traffic redirection to other instances.

When the user connects two routers in different instances, each instance registers the remote connection. When the network traffic passes through this special connection, the network emulator encapsulates the network packets into a new packet, storing extra information about the source router and the destination router.

The receiving instance, uses the added information to re-inject the network packet into the emulated network to continue its routing journey.

3.4.4 Network Topology

To allow traffic flow, routing is necessary. When the user connects two routers, the routers share information about their routing rules, providing information about the weight of each rule. This weight is utilized to enable the routers to only update their routing rules if the observed weight is lower than what is currently configured. This weight value is used to know the distance until the destination. For example, if a router has a routing rule with a weight value of zero, than that signals that the router is directly connected to the desired node.

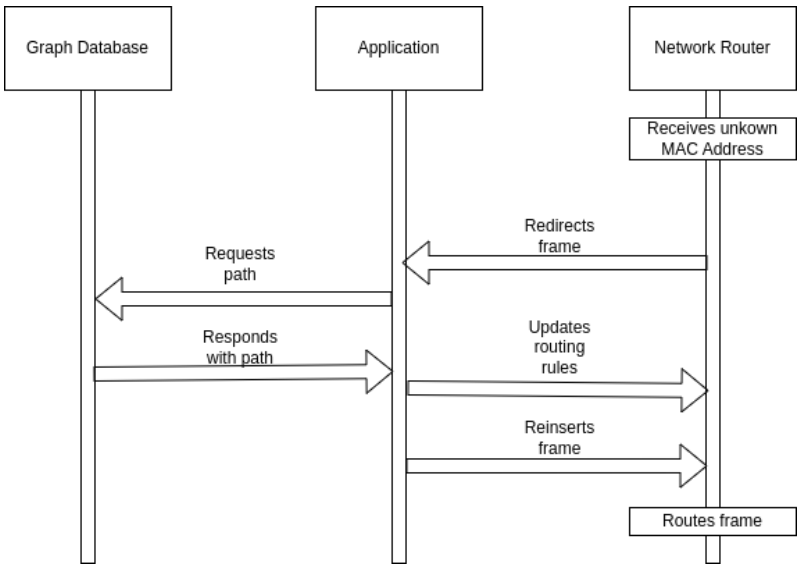


Figure 3.5: Application flow between GONE and Graph Database to obtain a MAC address

However, with the addition of a new container to the emulation, the routers do not automatically reflect this change. As described earlier, when a router does not know the destination of a network packet, it requests GONE for an answer. The emulation calculates the shortest path between the asking router and the desired router, and updates all the affected routers with new routing rules, allowing the network traffic to reach its destination. Fig. 3.5 shows the steps taken to obtain a new routing rule for an unknown destination.

3.4.5 Traffic Interception

To allow for traffic shaping, we must intercept the traffic between containers in the network, which allows us to, for instance, introduce arbitrary delays, drop packets, amongst others. The traffic is intercepted by GONE-Proxy.

By creating a specific docker network for the emulation, a new network namespace is created, allowing containers in the same network to communicate with each other. By leveraging this configuration that is already handled by docker, the proxy can intercept the network traffic of every container present by using eBPF and XDP.

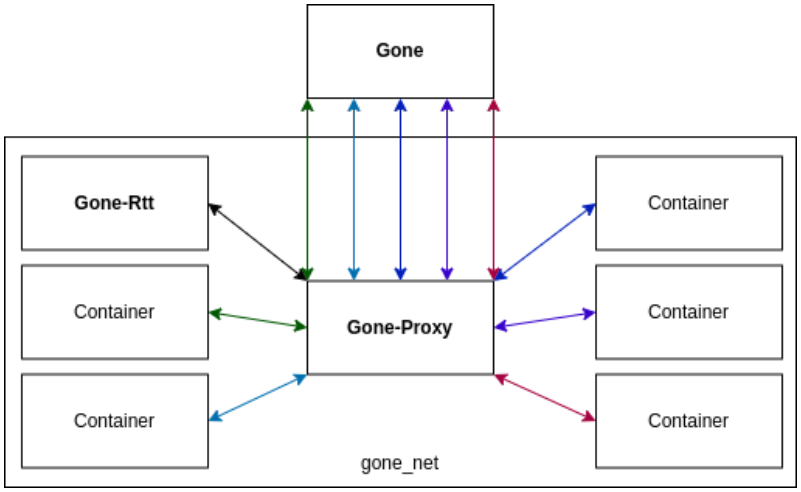


Figure 3.6: Application deployment inside of docker network namespace

For each container present in the network, the proxy creates a socket for each specific container, which GONE connects to, receiving network traffic from that particular container. Fig. 3.6 presents a diagram of the communication patterns between the proxy and the emulation. In order to apply the correct traffic shaping, proxy registers the received time of every packet. GONE uses this registered time to apply the configured network properties.

3.4.6 Reducing Kernel Latency

Due to the approach chosen to design the network emulation, requiring the interception of the network traffic from a given container, naturally there is a delay introduced to the network traffic until it reaches the proxy, as well as an extra delay from the proxy to a given application. To reduce the latency due to processing overhead, GONE-RTT was developed.

GONE-RTT is a simple program that configures a UDP socket, and awaits for requests from the proxy. Fig. 3.8 shows the JSON body message sent from the proxy.

Listing 3.8: JSON body of the round-trip time message calculation between Gone-Proxy and Gone-RTT

```
{
  "StartTime": "2024-12-08T14:41:58.965969856Z",
  "ReceiveTime": "0001-01-01T00:00:00Z",
  "TransmitTime": "0001-01-01T00:00:00Z",
  "EndTime": "0001-01-01T00:00:00Z"
}
```

To discover the minimum delay between an application and the emulator, GONE-RTT and GONE-Proxy exchange a configurable number of messages in order to measure latency. The proxy starts by registering the current time, `StartTime`, and sends it to GONE-RTT. GONE-RTT registers the received time in `ReceiveTime` field of the message and starts a

	Throughput (Gbits/sec)
With TCP Checksum Offload	49.2
No TCP Checksum Offload	7.62

Table 3.1: Throughput measurements with and without TCP Checksum Offload

new measurement in `TransmitTime`. After GONE-Proxy receives the message, it stores the current time in `EndTime`.

$$\Delta_{receive} = ReceiveTime - StartTime \tag{3.1}$$

$$\Delta_{transmit} = EndTime - TransmitTime \tag{3.2}$$

With this information, it is possible to discover the extra delay introduce by calculating the difference between the observed times, as shown in Equations 3.1 and 3.2.

To pass this information, the proxy starts a socket and publishes the results. The emulation, connects to this socket, reads the value and then applies the delay to reduce the difference between the configured delay and the observed delay during emulation.

3.5 Limitations

Through the development of this work, a limitation has been identified. For example, in order to successfully deliver network traffic to an application, it was necessary to change some network interface settings in every container.

By analyzing communication between containers, using TCPDUMP, we noticed that checksums were flagged as incorrect, but the packets were still delivered to the application. This is because the checksum calculation is offloaded to the network interface card (NIC), and calculated only when the packet leaves the network; it is not needed for local traffic. However, when we intercept the traffic, the incorrect checksum actually stops the packets from being delivered.

We conjecture that this happens because in the former scenario, the network traffic contains extra information that signals as local traffic, thus not requiring checksum validation. In our approach, the interception of network traffic leads to the loss of that context, and thus when reintroducing it to its final destination, the Linux kernel treats it as foreign traffic, requiring checksum validation. To circumvent this, we need to calculate the correct checksum. Since we can't rely on hardware, because the NIC is not involved in this communication, the only option is to calculate with software–turning off checksum offloading. To observe the difference in performance, we executed an iperf3 test between two containers with and without TCP checksum offload in a machine with TCP offloading capabilities. Table 3.1 shows the results obtained, where we see that it dramatically lowers throughput.

Due to design choices, there are several limitations imposed in the available operations of the network emulator. For instance, the user is not capable of connecting nodes to bridges created on different machines, to reduce cross communication between instances. The same is applied to connections between bridges and routers.

Another limitation of GONE is that it cannot handle restarting containers. When a container fails and restarts, docker creates new network interfaces, thus requiring a new bootstrap for the container. Since GONE indirectly leverages docker, it cannot perceive when a docker container as failed and restarted. Another limitation regarding containers is that when trying to add a new application to the emulation, the container will execute briefly, until GONE sends the pause command to Docker. In this scenario, containers that start by trying to request something, i.e., make a request to a server, typically fail due to not being able to contact the server.

3.6 Summary

In this chapter, we gave a thorough explanation of GONE and its components, providing a description of the available operations and detailing the internal logic that defines our work.

We started off by first recalling motivation for this work, followed by defining the requirements for this work. Next we presented our contribution, GONE, providing a detailed description of the architecture of the system and explaining the operation of each subsystem, detailing GONE-RTT, a program that allows to bootstrap the docker network and is used to measure the RTT between the emulation and an application in conjunction with GONE-Proxy, a component that intercepts network traffic and redirects it to the emulation.

After a brief description of the GONE environment, we detail how to configure and deploy a GONE instance, detailing the necessary dependencies, namely docker and a graph database, while also presenting the available environment variables the user can resort to configure each subsystem.

Following the steps of configuring and deploying an experiment, we detail the internal logic of GONE. First, we detail how to add a new application to the emulation, requiring some synchronization steps between GONE and GONE-Proxy.

Next, we explained how the network emulation is implemented, detailing how the programming language is leveraged to implement the relevant data structures and apply the desired network properties. To complement the emulation, we detail the behavior of the routers and how they are capable of configuring their routing tables through sharing information about their connections or requesting the application for routing rules. Lastly, we explain how GONE manages its instances, by following a Leader-follower approach, distributing the operations to the relevant instance and how each instance redirects network traffic to other instances.

We finish this chapter by addressing some limitations encountered or which were consequences of the design choices.

In the following chapter, we present the evaluation done to validate this work, presenting some micro-benchmarks about the performance of GONE and its operations against another network emulator, and some macro-benchmarks, evaluating the capability our work to emulate distributed systems in large network topologies.

4

Evaluation

In this chapter, we present the evaluation of our work, that is focused on accessing its capabilities of emulating a network topology and its features.

We start this chapter by presenting the objectives of our evaluation, detailing the goals for our work. Next, we present the evaluation done to validate our work. In more detail, we evaluate how our network emulator implements the desired network properties, and how these are influenced by a network distributed across many instances on different physical machines, followed by testing done to evaluate the performance of our network emulator in regard to processing network traffic or large scale topologies.

After evaluating the network properties, we demonstrate the existing operations the user can leverage to modify a running network topology. We finish this chapter by discussing operations that allow the user to further extend the capabilities of our network emulator by implementing custom network operations.

4.1 Evaluation plan

To evaluate this work and validate the requirements defined in Chapter 3, we evaluate our work in three distinct dimensions, that we define as follows:

- **Network Emulation:** We evaluate our work in regard to the desired network properties by conducting various tests to evaluate each network property, testing locally and in a distributed setting. Furthermore, we also deploy relevant network scenarios to evaluate how our network emulator behaves when dealing with large network topologies.

- **Dynamic Experiments:** Besides network properties, our network emulator allows modifying the network properties at runtime while an experiment is running. To demonstrate such events, we deploy network scenarios that showcase new network behavior and evaluate how these events affect the experiment being executed.

- **Extensibility:** As to increase the available network operations the network emulator has, we demonstrate, using three uses cases, its capability of extending the network emulator with new network operations.

4.1.1 Ping and Iperf3

We leverage the usage of the Ping tool to observe the latency between two nodes in our emulated network. The Ping tool provides information regarding the observed latency between two communicating hosts. This tool will be used throughout many experiments as it allows observing not only the configured network properties as well as observe new events in the experiment.

The usage of Iperf3 allows the measurement of the configured capacity of a given link, providing insights whether the network emulator is capable of constraining the generated network traffic to the configured transmission rate. It is also capable of demonstrating different behavior when modifying the network topology.

4.1.2 Environment

All the tests and deployments conducted to evaluate our work were done on the DI Cluster, utilizing the "Moltres"cluster, consisting of 10 machines, where each machine has 2 x AMD EPYC 7343 CPUs and contains 128 GiB DDR4 3200 MHz of memory, interconnected by a 20Gbit ethernet network.

4.2 Network Emulation

In this section, we present the results of the experiments conducted to test the internals of our network emulator and discuss the obtained results.

The following sections detail the experiments that were conducted for every relevant network property: latency, bandwidth, jitter, packet loss, and link congestion. Furthermore, we test the processing capabilities of our network emulator, evaluating the overhead of emulating network links.

4.2.1 Latency

To observe the configured latency is applied in the emulation, a simple network was created, consisting of two nodes, connected to a bridge (see Fig. 4.1). Next, the properties of the link were modified through the various experiments, changing the values of latency to 10, 20, 50, 100, 250, 500, and 1000 milliseconds of delay. The same network was deployed in Kollaps in order to compare its values with the observed values from GONE.

The test consists of 10000 ping requests between the client and the server with an interval between ping messages of 5 milliseconds each. This interval of messages was chosen for a more accurate observation of the configured latency.

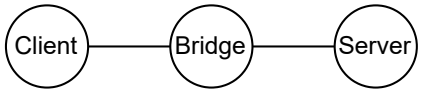


Figure 4.1: Simple network topology containing one server and client connected to a bridge

Latency	GONE			Kollaps		
	Minimum	Average	Error	Minimum	Average	Error
10	10.437	11.248	12.48%	10.043	10.079	0.79%
20	20.562	21.210	6.05%	20.042	20.404	2.02%
50	50.189	51.315	2.63%	50.056	50.315	0.63%
100	100.248	101,190	1.19%	100.050	100.367	0.37%
250	250.154	251.217	0.49%	250.035	250.301	0.12%
500	500.264	501.166	0.23%	500.044	500.340	0.68%
1000	1000.140	1001.223	0.12%	1000.045	1000.360	0.036%

Table 4.1: Latency measurements (milliseconds) in a local setting for GONE and Kollaps

Table 4.1 shows the observed ping measurements comparing Kollaps with GONE regarding the configured network topology. As it is observed, our work exhibits a higher latency average than Kollaps, adding an extra millisecond of latency than expected. This is due to the nature of the chosen architecture, requiring the transfer of network traffic from kernel space to user-space through XDP, and thus introducing extra latency that cannot be accounted for. It should also be noted that this extra overhead remains stable, and as such could be mitigated by decreasing it from the introduced latency. However, verifying that the overhead is constant across different setups was not done, so we decided not to do this.

Now, we also evaluate the accuracy of the configured latency in a distributed setting. Due to the nature of GONE that emulates every network component, there is added overhead when communication between two instances occur. We leverage the previous scenario, but with extra components to allow for cross traffic between machines, as shown in Fig. 4.2, deploying two extra routers and a bridge.

Since GONE allows for 0 ms latency links, we configure the intermediate links with 0 latency and only change the latency of the links that connect the nodes to the bridges. For Kollaps we use the same network topology as shown in Fig. 4.1 to test latency locally, since Kollaps is able to do so.

Table 4.2 shows the results of the accuracy of the configured latency in the configured network, and increasing the latency configured in the mentioned links.

As shown, our network emulator is capable of maintaining the same latency accuracy whether the network is distributed or deployed locally. Our network emulator is capable of taking in consideration the added overhead of transferring network traffic between machines and adjust the latency applied.

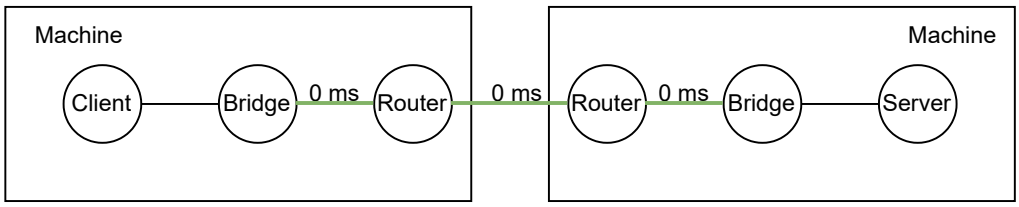


Figure 4.2: Simple network topology containing one server and client deployed in different machines

Latency	GONE			Kollaps		
	Minimum	Average	Error	Minimum	Average	Error
10	10.241	11.143	11.43%	10.092	10.372	3.72%
20	20.118	21.243	6,22%	20.088	20.434	2.17%
50	50.099	51.216	2,43%	50.167	50.771	1.54%
100	100.090	101.211	1,21%	100.158	100.456	0.46%
250	250.046	251.127	0,45%	250.118	250.498	0.20%
500	500.087	501.177	0,24%	500.275	500.747	0.15%
1	1000.108	1001.200	0,12%	1000.160	1000.647	0.07%

Table 4.2: Latency measurements (milliseconds) in a distributed setting for GONE and Kollaps

4.2.2 Bandwidth

To evaluate the configured bandwidth, we leverage the Iperf3 tool. We leverage the previous network topologies for testing latency to evaluate bandwidth, locally and across instances.

However, in this scenario the links are configured with a latency of 10 milliseconds each, while we modify the transmission rate of the links between nodes and the bridge. We do the same network topology for Kollaps. For testing, we vary the bandwidth for the respective values of 512 Kbps, 1 Mbps, 10 Mbps, 100 Mpbs, 250 Mbps, 500 Mbps, 1 Gbps, 2 Gbps.

Bandwidth	GONE	Kollaps
512 Kbps	419 Kbits/sec	457 Kbits/sec
1 Mbps	834 Kbits/sec	892 Kbits/sec
10 Mbps	9.40 Mbits/sec	9.53 Mbits/sec
50 Mbps	47.4 Mbits/sec	47.6 Mbits/sec
100 Mbps	95.0 Mbits/sec	95.2 Mbits/sec
250 Mbps	234 Mbits/sec	238 Mbits/sec
500 Mbps	473 Mbits/sec	475 Mbits/sec
1 Gbps	912 Mbits/sec	949 Mbits/sec
2 Gbps	1.10 Gbits/sec	1.89 Gbits/sec

Table 4.3: Iperf3 measurements observed locally for Kollaps and GONE

As shown in Table 4.3, GONE and Kollaps show very close values regarding bandwidth

limits up until 1 Gbps. After 1 Gbps, it is clear that GONE starts to limit the network traffic to 1.10 Gbits/sec. This is a clear limitation of our approach since it requires receiving network traffic from the kernel and transfer it to the network emulation, thus limiting the transmission rate, influencing the observed latency by Iperf3.

To test bandwidth limits across multiple instances, we deploy the network topology shown in Fig. 4.2, configuring the node links with a 10-millisecond delay, and vary the transmission rate. Table 4.4 shows the results of executing Iperf3 with various rate configurations for Kollaps and GONE.

Bandwidth	GONE	Kollaps
512 Kbps	419 Kbits/sec	457 Kbits/sec
1 Mbps	838 Kbits/sec	892 Kbits/sec
10 Mbps	9.43 Mbits/sec	9.53 Mbits/sec
50 Mbps	47.5 Mbits/sec	47.6 Mbits/sec
100 Mbps	94.8 Mbits/sec	95.1 Mbits/sec
250 Mbps	237 Mbits/sec	238 Mbits/sec
500 Mbps	468 Mbits/sec	475 Mbits/sec
1 Gbps	714 Mbits/sec	948 Mbits/sec
2 Gbps	738 Mbits/sec	1.89 Gbits/sec

Table 4.4: Iperf3 measurements observed in a distributed setting for Kollaps and GONE

As Table 4.4 shows, Kollaps and GONE demonstrate similar bandwidth constraints until the 1 Gbps restriction, where our network emulator reaches its limits sooner compared to the previous testing, due to extra processing for the network traffic to the reach the other instance, requiring the encapsulation of the network traffic with extra data to allow the other instance to introduce the traffic into the correct component in the emulated network.

4.2.3 Bandwidth Congestion

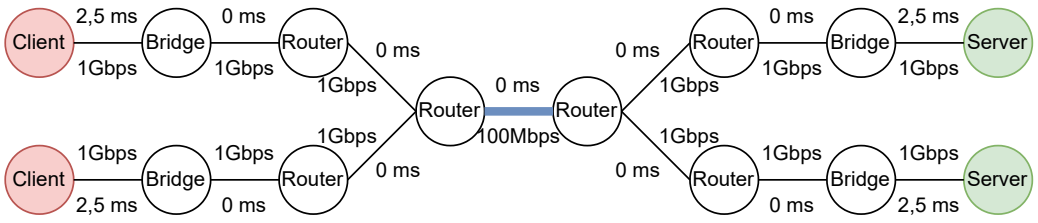
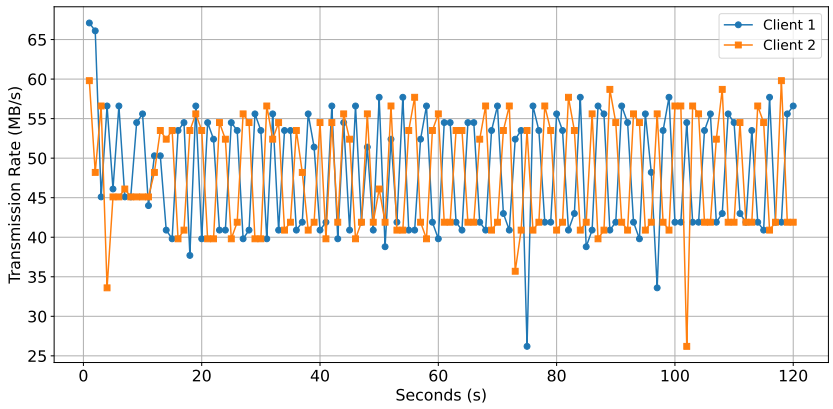


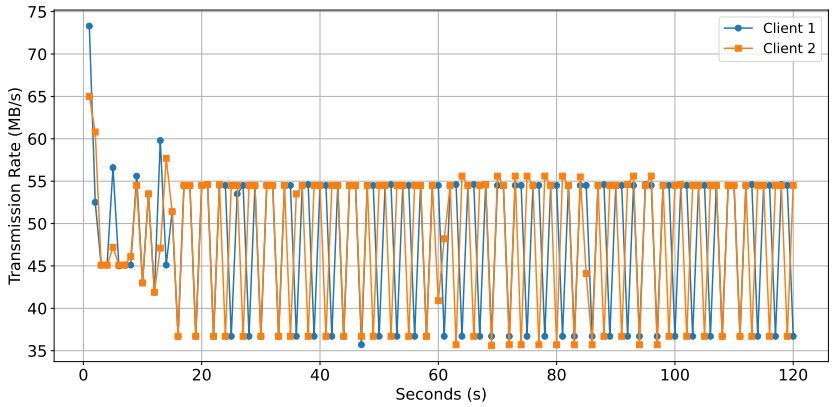
Figure 4.3: Emulated Network topology containing 2 clients and 2 Iperf3 servers

To observe how applications are affected by sharing the same network, we devised two tests. The first test evaluates how the transmission rate between two clients gets affected by sharing the same network link, while the second test observes how each application changes when introducing new clients to the network.

For the first test, we deployed 4 nodes, 2 clients and 2 iperf3 servers, as shown in Fig. 4.3, where the shared link is restricted to 100Mbps. The test consists of executing an Iperf3 test for 120 seconds. We executed the same test on Kollaps for comparison.



(a) Transmission Rate over Time per Client (GONE)



(b) Transmission Rate over Time per Client (Kollaps)

Figure 4.4: Transmission Rate per Client

Fig. 4.4 shows the transmission rate per second for each client over a period of 120 seconds for each network emulator. During execution, each client adapts its own transmission rate depending on the observed available link bandwidth, where each client reaches a steadying bitrate of 50Mbps. As observed, both clients present similar behavior when executing the Iperf3 test in different behaviors.

However, Kollaps, emulates link congestion by restricting the observed transmission rate of each client, adapting the network properties to emulate link usage. GONE on the other hand, since it intercepts network traffic, emulates link congestion by passing network traffic through the same (emulated) link and as such exhibits results that are slightly more stable than Kollaps that has throughput dropping to close to instant multiple times.

To demonstrate this difference in handling link congestion, another test was devised. In this scenario, we increase the number of clients and servers to 4, executing the Iperf3 test in each client at different times. Fig. 4.5 shows the deployed network for both network

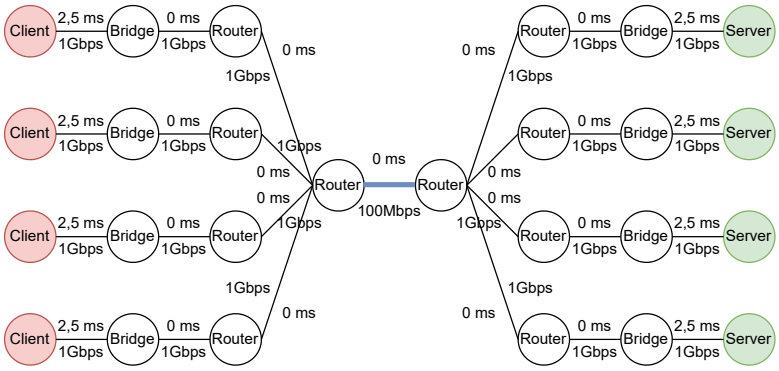
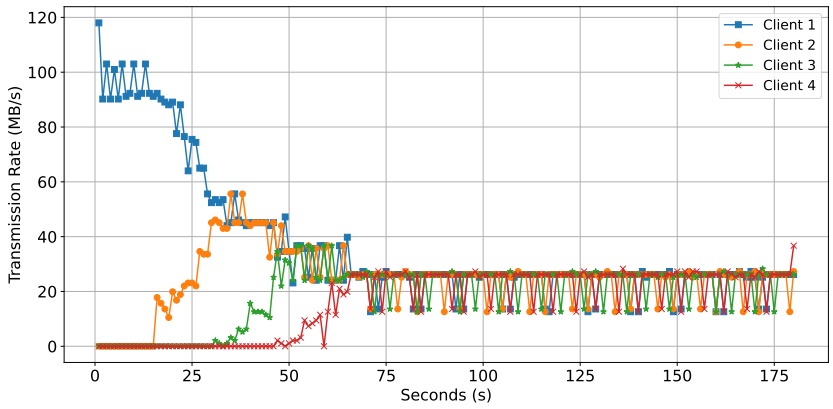
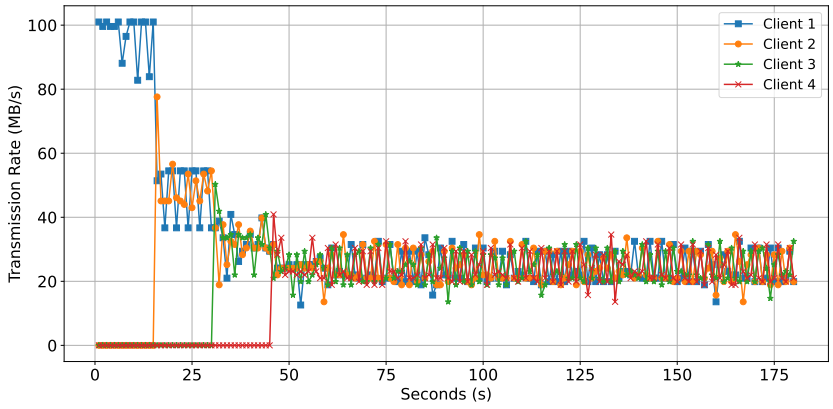


Figure 4.5: Network topology containing 4 clients and 4 Iperf3 servers

emulators. Also in this network, the shared link is restricted to 100Mbps as well.



(a) Transmission Rate per Client (GONE)



(b) Transmission Rate per Client (Kollaps)

Figure 4.6: Transmission Rate over Time per Client

Fig. 4.6 shows the transmission rate for each client per network emulator. Different to what was previously observed for two clients, where they quickly adapted their transmission rate, in this scenario, executing a new Iperf3 test at a latter time when the current clients reach a stable transmission rate shows each client adapting its transmission rate

over a period of time, until reaching equal sharing of the network link across all clients, due to better observability of the link usage. Kollaps, on the other hand, to emulate link congestion, monitors the usage of the network link and restricts the available bandwidth for each network flow to an equal share by dropping network packets.

4.2.4 Jitter

To demonstrate how our network emulator applies jitter to a given link, we deployed the same network as in Fig. 4.1 with 50 ms of latency between client and server. To obtain a baseline, we executed a ping test with 10000 requests. Next, we configured a jitter value of 12,5 ms between client and server and generated the same 10000 requests. For comparison, we deployed the same network in Kollaps. Fig. 4.7 shows the latency of each request for each network emulator.

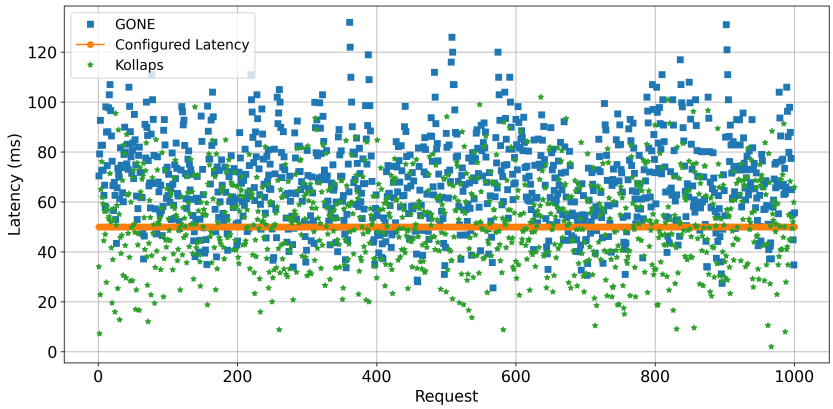


Figure 4.7: Latency per Request

As observed, the applied jitter differs from each other. Both network emulators assume the Normal Distribution for jitter values where the mean and standard deviation correspond to the link latency and jitter value, respectively for a given network link. However, Kollaps, due to reducing the network properties between hosts, calculates the jitter value differently. In our network emulator, the jitter only applies to the network link that it is configured on, allowing finer control over the network properties.

4.2.5 Packet Loss

To evaluate how our network emulator applies packet loss, we used the same network scenario used for evaluating jitter, with a configured 10% drop rate between client and server. To observe packet loss, we executed five iperf3 tests generating UDP network traffic, allowing the execution of the test in a single direction. Next, execute 5 ping tests with 10000 requests each. The same test was done in Kollaps for comparison.

Table 4.5, shows the results obtained for the five tests for each tool, presenting similar results for both network emulators. As observed with Iperf3, the drop rate reflects the configured drop rate. In contrast, sending ping requests shows a much higher drop rate,

	Tool	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Kollaps	Iperf3	9,93%	9,98%	10,00%	9,94%	10,06%	9,98%
	Ping	19,40%	19,57%	19,41%	19,47%	19,62%	19.49%
GONE	Iperf3	10,00%	10,00%	10,28%	10,17%	9,96%	10,08%
	Ping	18,82%	19,13%	18,95%	19,06%	19,06%	19,00%

Table 4.5: Packet Loss for Iperf3 and Ping

due to applying the same properties for both directions of the network link. This lets us conclude that both network emulators apply the same packet loss. In case of ping, if the user only wishes to apply packet loss to only one direction, Kollaps allows unidirectional links, but this leads to the requests to follow a different routing path. However, our network emulator allows the capability applying packet loss to only one direction but still use the same network link. Section 4.4 provides an example where the user can leverage traffic interception to extend the functionality of our network emulator and only apply drop rate to a single direction.

4.2.6 Scalability

Besides making sure that our network emulator correctly applies the network model being configured, it is also important to evaluate its performance when emulating large networks. To evaluate our network emulator, we leverage Ping and Iperf3 to measure latency and bandwidth, and deployed various networks, consisting of two hosts, but varying the number of links between them. Fig. 4.8, shows the used topology. For each scenario, the number of links created was 10, 20, 50, 100, 200, 500, 1000, 5000, 10000. Every link in the network was configured with 0 latency and 10Gbps of available bandwidth. We tested this networks in a single instance and distributed across 10 machines in the Moltres cluster.

Figure 4.8: Network Topology with varying routers between Client and Server

Fig. 4.9 shows the average latency between client and server for a local deployment and a distributed deployment. In this scenario, we configured the network links with 0 latency to demonstrate the overhead of the emulation. As it is shown, the average latency observed for a single instance is lower than a distributed setting, due to the added cost of redirecting network traffic across machines.

Regarding traffic processing, we conducted an Iperf3 test for 60 seconds, generating 1Gbps of udp network traffic in both directions, since udp traffic is not influenced by the latency observed, as opposed to tcp.

Fig. 4.10a shows the transmission rate and receiver rate for the client for each network scenario in a single instance.

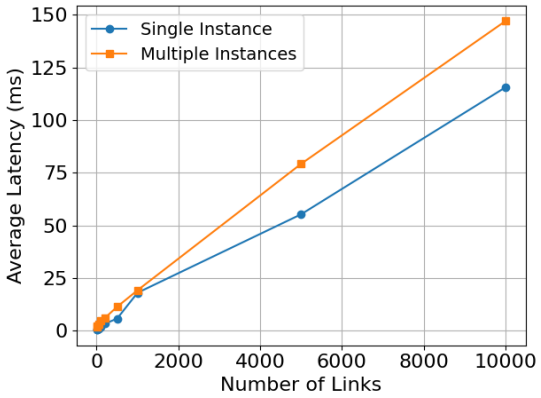
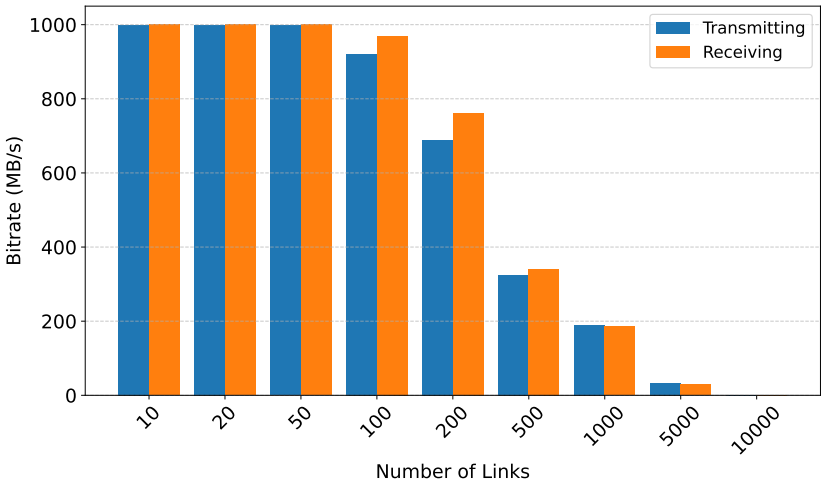
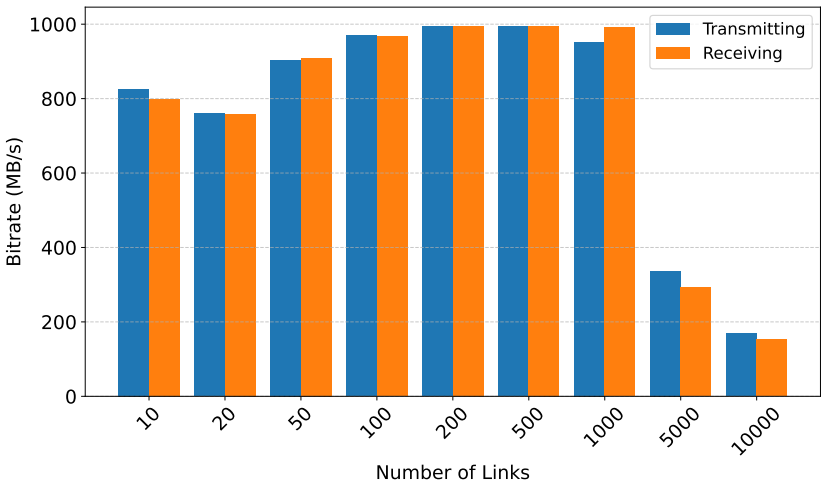


Figure 4.9: Average Latency per Number of Links



(a) Transmission rate per Link configuration (single instance)



(b) Transmission rate per Link configuration (multiple instances)

Figure 4.10: Iperf3 test scenarios locally (a) and distributed (b)

As observed, our network emulator correctly handles the generated network traffic up until 200 links, where we start to see a degradation in performance. This drop in performance is due to the network emulator not being able to manage too many active links, and thus, the internal queues get full and start to drop network packets.

The drop in performance can be mitigated by deploying the network topology across multiple instances, as shown in Fig. 4.10b.

Fig. 4.10b shows our network emulator handling network traffic up until the 1000 network links, reducing performance for 5000 links and 10000 links. This leads us to conclude that our network emulator is able to scale the network emulation across multiple machines. Kollaps on the other hand, is limited to emulating at most 65535 network links, and thus is limited in the size of network topologies it can emulate.

4.2.7 Large Topologies

Another important aspect to evaluate in our network emulator is its capability of emulating large topologies. For that effort we conducted two different tests: we deploy a simple benchmark consisting of varying hosts, and for each network topology, execute a workload of ping requests to random hosts in the network; we deploy IPFS [104], a file sharing peer-to-peer network.

4.2.7.1 Large Topologies: Ping

For this test, we deploy network topologies consisting of 10, 20, 50, 100, 250, 500 hosts, with each host having a direct connection to every other host. The experiment consists for each network, of a given application sending ping requests to random nodes in the network, for a total of 15 minutes. The configured latency between any two hosts is 10 milliseconds. We deploy the same experiment in Kollaps for comparison. The experiments where deployed in a distributed way through all Moltres machines.

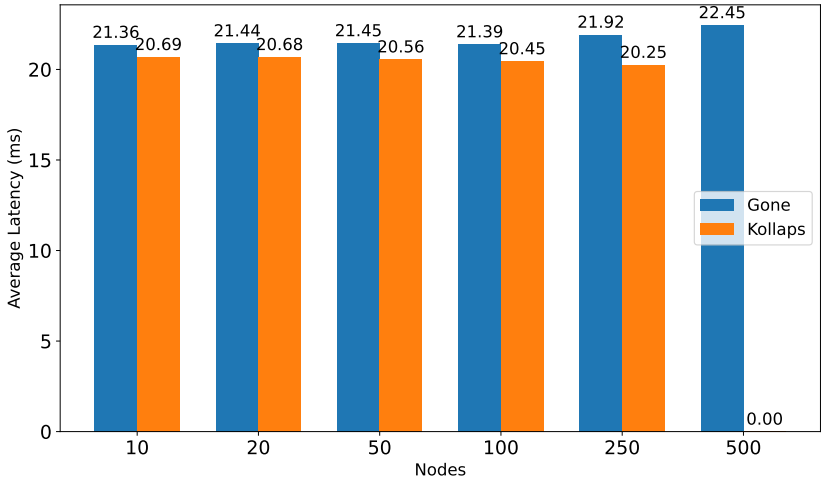


Figure 4.11: Average Latency per configured network topology

Kollaps	GONE
134.316	76.515

Table 4.6: Average latency (milliseconds) for fetching a file

Fig. 4.11 shows the average latency for each network topology. As presented, both network emulators, present an average latency close to 20 milliseconds for each experiment. Our network emulator presents a slightly higher latency, an extra millisecond, when compared to Kollaps. However, for experiments containing more than 65535 network links, Kollaps is not capable of configuring experiments, failing during setup time, due to leveraging Linux TC for emulation. Our network emulator, does not use Linux TC and as such is capable of running much larger experiments, presenting an observed average latency of 22 milliseconds. This test lets us conclude that our network emulator is capable of deploying larger network topologies and is not restricted to 65535 network links.

4.2.7.2 Large Topologies: IPFS

To validate our solution in its capability of deploying a large distributed system, we chose to deploy an IPFS network evaluate whether our network emulator is capable of deploying large network topologies.

The choice for deploying an IPFS network is due to its decentralized nature in file-sharing. IPFS allows you to publish or retrieve data from the network in a decentralized fashion. This means that when a user publishes a file to the IPFS network, the file is dispersed throughout the network in blocks of 512 or 1024 bytes, spread across multiple IPFS nodes. This approach leads to nodes in the network, when fetching a file, will contact multiple nodes to obtain a given file, where each contacted node provides a fragment of the desired file resulting in a random communication pattern.

We deployed a IPFS network consisting of 250 nodes, with every node connected to all the other nodes. The network was configured with a 15 ms of latency and 100Mbps of bandwidth between hosts. The simple test consists in fetching 1000 files from the network, and observe the average latency. We also deployed this network in Kollaps.

Table 4.6 shows the average latency obtained for Kollaps and GONE. As observed, the values differ by quite a lot. This is due to the randomness in publishing the files in the IPFS network. This simple test lets us conclude that our network emulator is capable of testing large topologies.

4.2.8 Summary

In this section we performed and reported on a series of testing scenarios to evaluate how our network implements network emulation, evaluating its emulation of latency and bandwidth, locally and distributed, and compared against similar network emulator, Kollaps, demonstrating that despite leveraging different technologies, our emulator provides a very close emulation experience compared to Kollaps. Furthermore, we also evaluate

how both network emulators emulate link congestion, demonstrating that our network emulator provides a much smoother experience when multiple applications are sharing the same link when compared to Kollaps, which instantly adapts the network link to share its bandwidth across all flows.

Besides evaluation network properties, we also evaluate the performance of our emulator, by creating network scenarios that force our network emulator to manage multiple active links. These scenarios demonstrate the capability of our network emulator of handling network traffic in the emulated network, demonstrating the added overhead of network traffic crossing many network links, and by distributing the emulation to many instances we are capable of processing more network traffic.

Lastly, we compare our network emulator against Kollaps when emulating large topologies, where we deployed fully connected network topologies, concluding that our network emulator remains accurate when compared to Kollaps, and is capable of emulating larger network topologies than Kollaps, due to not having the restriction of using Linux TC, limiting to the emulation to 65535 network links.

4.3 Experiments With Dynamic Network Scenarios

In this section, we demonstrate the existing network operations the user can leverage to create dynamic experiments with changing network conditions. We demonstrate the available network operations to modify a network topology during runtime, modifying the network properties of a link, changing the network topology by connecting routers during execution, or to simulate an unstable network by disabling temporarily a network link. We also present these network operations by deploying a popular distributed database, Cassandra, and deploy three clients that perform the YCSB benchmark. During deployment, we introduce network events, allowing us to observe how these network events affect the clients. We finish this section by also providing two examples of network disruption, a simple scenario of disabling a network link, and the emulation of link-flapping.

4.3.1 Changing Network Properties

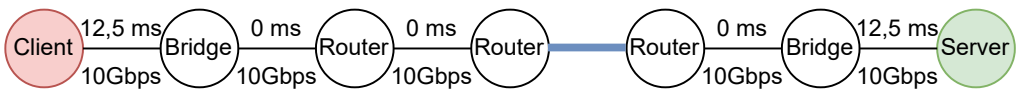


Figure 4.12: Simple network topology containing one server and client deployed in different machines

To demonstrate the network topology modifications that our network emulator is capable of, we resort to various network scenarios. Firstly, we deploy a simple network topology, shown in Fig. 4.12, containing a client and server, separated by three routers.

The client and server are configured with a 100Mbps bandwidth limit and a latency value of 25 milliseconds. Next, we execute a ping test, sending ping requests at an interval of 0.05 seconds for a total of 1000 requests while changing network properties.

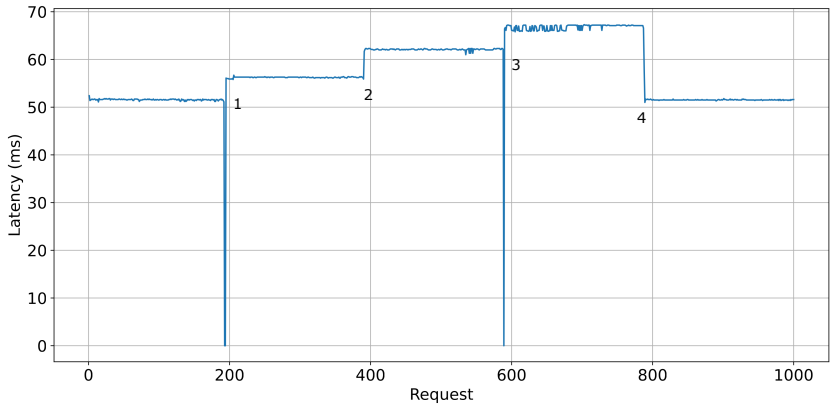


Figure 4.13: Latency per Request during network changes

Fig 4.13 reports the results of the experiment. Periodically, we change the network properties of one of the links between routers, slowly increasing the configured delay throughout the experiment, adding a latency of 5 milliseconds. As demonstrated, there is an increase in latency after changing the network properties. It is also worth noting that for each change, there is a loss of some requests, presented as 0 ms latency. This packet loss is due to changing the properties of a network link, where there is a brief moment where there is a disconnection between routers, leading to packet dropping due to the router not having any route configured for that request.

Another relevant change of network properties is changing the bandwidth of a network link. Using the previous network, we execute an Iperf3 test during 60 seconds, where we temporarily configure a network link with 10Mbps.

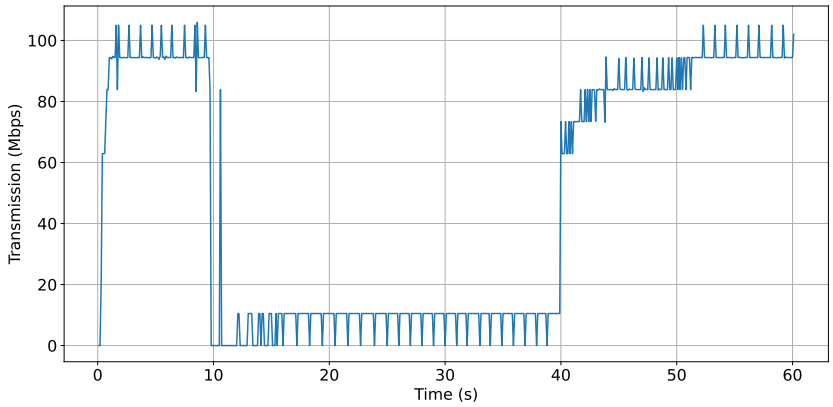


Figure 4.14: Transmission rate over Time during network changes

Fig. 4.14 shows the results of the Iperf3 test. As observed, initially, the Iperf3 application reports a transmission rate close to 100Mbps, as configured in the network. With a change

in network properties, changing the bandwidth to 10Mbps, the Iperf3 reports a drop in its transmission rate, corresponding to a temporary disconnect of the network. After the configuration of the new link, the Iperf3 reports a transmission rate equal to the link capability.

Returning the network link to its original properties, demonstrates how the Iperf3 manages the transmission rate when it identifies that it can transmit more data, slowly increasing its bitrate until reaching the link's configured bandwidth. This example shows how we can leverage our network emulator to study the behavior of an application when the network properties of a network link change during testing.

4.3.1.1 Changing Network Topology

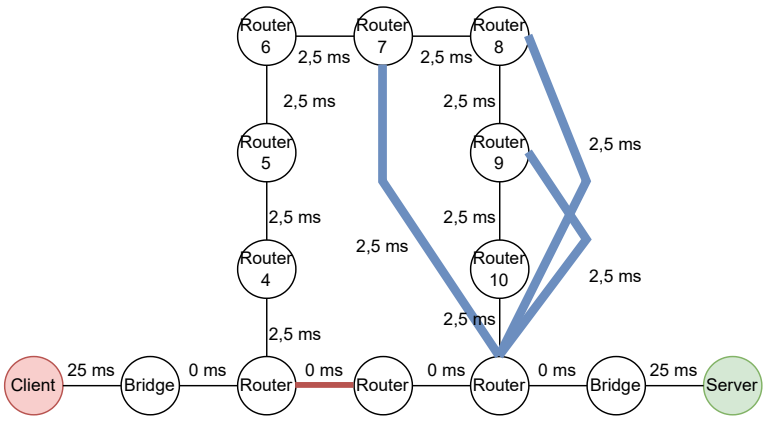


Figure 4.15: Network Topology used to demonstrate network changes. The red link represents a disconnect, while the blue links represent a new connection

An alternative to changing the network properties of a network link, is the ability to change connections between routers, by adding or removing them. Since our network emulator allows the user to change network topology at runtime, we conduct a simple test where we change the network properties by adding and removing network links. Fig. 4.15 shows the configured network. We execute a ping test to observe the latency of each request, and change the network topology, by disabling a network link, colored in red, forcing the network traffic to go another route. Throughout the test we make new connections, colored as blue. Each connection present contains a configured latency of 2.5 milliseconds for each direction.

Fig. 4.16 shows the results of this experiment. Initially, the observed latency is around 50 milliseconds, corresponding to the shortest path between the client and server. When disconnecting two routers (1), the network becomes "unstable", since it needs to reorganize to be able to find a new path until destination, leading to a higher latency for a particular request, since the router waits for a response of where it should send the packet. After setting the routing rules of the relevant router, the procedure continues, as shown by the

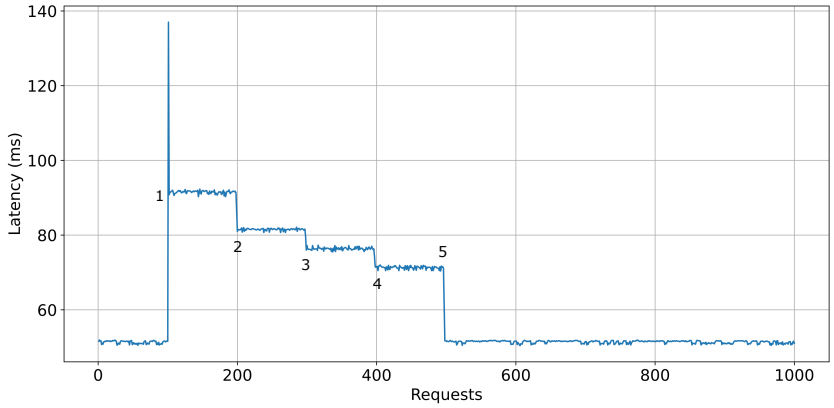


Figure 4.16: Latency per Request

increase in latency. The following changes to the network, (2, 3, 4 and 5) corresponds to the new connections made, reducing the observed latency.

However, the increase in latency that occurred in (1) does not occur in other network events due to our network emulator, after calculating the shortest path to a destination, automatically updates the affected routers. Furthermore, when connecting routers, they exchange information about their routing rules, and thus quickly configure their routing table without disrupting network flow.

4.3.1.2 Cassandra

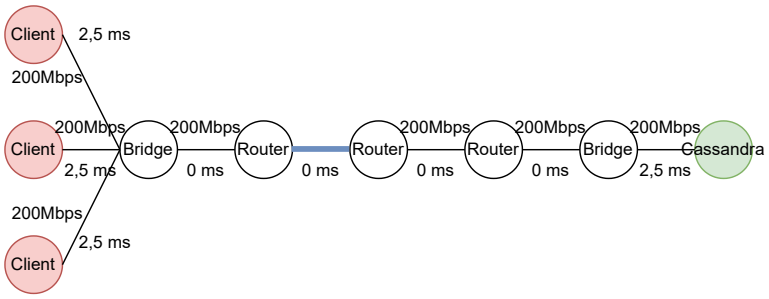


Figure 4.17: Cassandra Network Topology

Another perspective on leveraging network operations, is to observe how these network events influence the communication of a real application. As such, we deploy a simple network topology containing three clients and a single instance of a Cassandra database, as shown in Fig. 4.17. The clients and the database have a latency of 10 milliseconds, and are limited to 200Mbps. In this experiment, we change the bandwidth between routers, changing from 200Mbps to 100Mbps, 50Mbps, 40Mbps, 30Mbps, 20Mbps, and back to 200Mbps. Each client runs the YCSB [105] benchmark against the Cassandra [70] database, performing a workload of 50/50 read and write operations.

Fig. 4.18 displays the operations per seconds for each client over time. As observed,

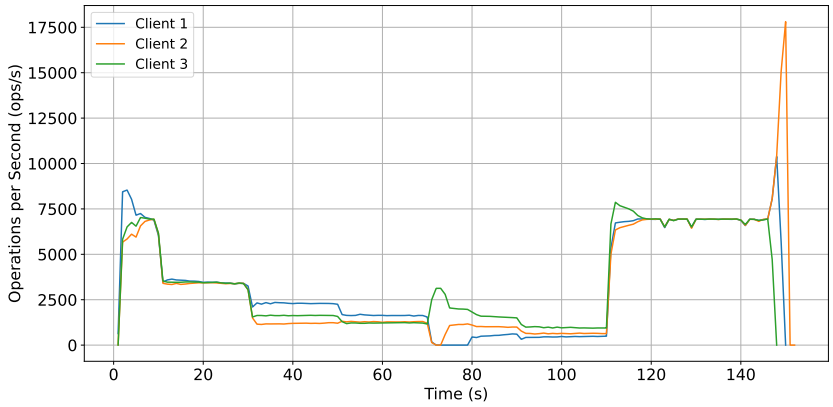


Figure 4.18: Operations per second over execution time

with each network change, the clients reduce the number of operations per second. It is also worth noting that for each change, all clients fight over the available bandwidth and slowly converge to an equal share of bandwidth, following the same behavior as shown previously in Section 4.2.3. This scenario allows us to observe how the number of operations per second is affected by the network changes made throughout the experiment.

4.3.2 Network Disruption

Besides changing network topology, GONE also allows for network disruption, temporarily disabling links, routers or bridges. This allows the user to disrupt the network flow without having to instantly reflect the action on the network. To demonstrate such operation, we leverage the simple network present in Fig. 4.1.

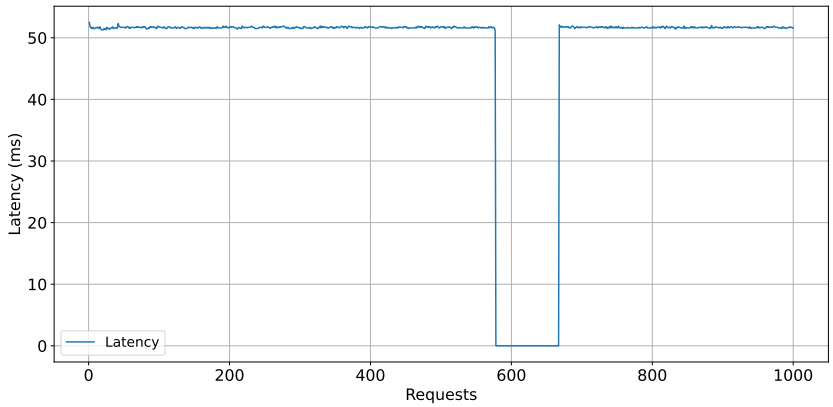


Figure 4.19: Latency per Request

Fig. 4.19 demonstrates a simple example of disrupting the network flow between two hosts. The requests with 0 ms latency correspond to failed ping requests.

The user can further extend this behavior and emulate other network events, for example link-flapping. We use the network topology shown in Fig. 4.12 and leverage the ping tool to demonstrate this event.

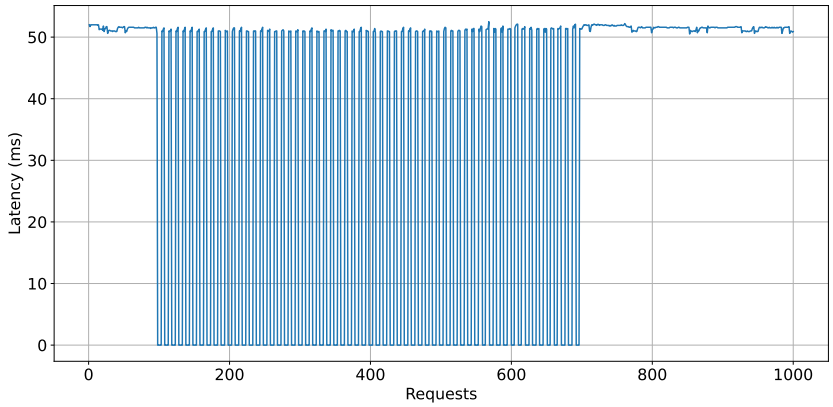


Figure 4.20: Latency per Request

Fig. 4.20 shows the occurrence of link-flapping. By quickly disrupting a network link for a brief period of 125 milliseconds, there is a loss of ping requests during this event.

4.3.3 Summary

In this section, we presented the available network operations in our network emulator that enable to change the network model at runtime. We demonstrated the ability to change network properties during runtime, change network topology by connecting or disconnecting components. Through Ping and Iperf3, we obtained relevant metrics that demonstrate these operations and how they affect an application. To complement this observation, we deployed a simple workload consisting of a Cassandra database and three clients, which executed the YCSB benchmark, demonstrating how network events affect the operations per second done by each client.

To conclude this section, we also displayed disabling network components in the emulation, to emulate a temporary network fault, and even leveraged the disruption operation to emulate link-flapping.

By not following an experiment based approach but instead providing an interactive environment, this allows the user to leverage existing operations to implement network operations that are relevant to the experiment. Furthermore, depending on how an application behaves, the user can introduce network events accordingly. In the next section, we present novel operations the user can leverage to further test an application.

4.4 Extensibility

GONE offers two novel features that allow the user to extend the capability of our network emulator. The user can further extend the network operations of our network emulator by either receiving a copy of the network traffic of a particular link, or hijacking the network traffic. In this section we present an example of leveraging the *Sniff* operation of our network emulator, by converting the received network traffic into the PCAP format

and redirecting it to a popular network debugging tool, tcpdump, and two examples using the *Intercept* operation, where it is possible to change the default network properties of a particular link, in this example are changing the latency for a particular IP address and applying packet loss to only one direction of a network link.

4.4.1 Sniffing

The user can convert a network link, into a sniffer link, receiving a copy of the network traffic that crosses that link. This allows the user to perform traffic analysis of a particular link. One example of leveraging this feature is the ability to observe the network traffic through tcpdump, a popular network tool.

To externalize network traffic and redirect it to tcpdump, we deployed a simple network topology containing 10 nodes, where each node sends icmp requests to every other node in the network. To convert the network traffic to the PCAP format, a custom application was developed, as shown in Annex II.2. This program connects to a socket created by our network emulator and converts it to the appropriate format. Fig. 4.21 shows the output of tcpdump when redirecting the output of the custom application onto tcpdump.

```
22:39:27.301409 IP (tos 0x0, ttl 64, id 26084, offset 0, flags [none], proto ICMP (1),
    length 84)
   10.1.0.101 > 10.1.0.100: ICMP echo reply, id 360, seq 198, length 64
22:39:27.301430 IP (tos 0x0, ttl 64, id 8776, offset 0, flags [none], proto ICMP (1),
    length 84)
   10.1.0.105 > 10.1.0.100: ICMP echo reply, id 396, seq 197, length 64
22:39:27.324353 IP (tos 0x0, ttl 64, id 38148, offset 0, flags [DF], proto ICMP (1),
    length 84)
   10.1.0.106 > 10.1.0.100: ICMP echo request, id 362, seq 198, length 64
22:39:27.324519 IP (tos 0x0, ttl 64, id 6763, offset 0, flags [none], proto ICMP (1),
    length 84)
   10.1.0.100 > 10.1.0.106: ICMP echo reply, id 362, seq 198, length 64
22:39:27.356313 IP (tos 0x0, ttl 64, id 30878, offset 0, flags [DF], proto ICMP (1),
    length 84)
   10.1.0.108 > 10.1.0.100: ICMP echo request, id 365, seq 198, length 64
22:39:27.356419 IP (tos 0x0, ttl 64, id 2252, offset 0, flags [none], proto ICMP (1),
    length 84)
   10.1.0.100 > 10.1.0.108: ICMP echo reply, id 365, seq 198, length 64
22:39:27.428353 IP (tos 0x0, ttl 64, id 6798, offset 0, flags [DF], proto ICMP (1), length
     84)
   10.1.0.100 > 10.1.0.106: ICMP echo request, id 405, seq 197, length 64
```

Figure 4.21: Output of tcpdump of a network link

This is a simple yet effective example that demonstrates the usability of having full access to the network traffic of our network emulator, allowing the user the creation of new approaches to testing distributed applications that are not covered by our network emulator.

Tests	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Improved Drop Rate	9,99%	9,98%	9,92%	9,90%	10,18%	9,94%
Old Drop Rate	18,82%	19,13%	18,95%	19,06%	19,06%	19,00%

Table 4.7: Default Drop Rate vs Improved Drop Rate

4.4.1.1 Changing Drop Rate

As shown in Section 4.2.5, when configuring the drop rate of a particular link, our network emulator applies half of the drop rate for both directions, which can skew results for bidirectional communication. To demonstrate the ability to only apply packet loss to one direction, we resort to the intercept operation to redirect traffic to an external application. Annex II shows the custom program that applies 10% packet loss.

Table 4.7 shows the results obtained when applying packet loss to only one direction of the chosen network link. This shows the capability of extending the network emulator with new operations.

4.4.2 Fine-grained Latency Injection

Another example of extending the network operations, is the possibility of applying different latency values to a given network flow across applications. In this example, we deployed the same network as presented in Fig. 4.5, with a configured latency of 20 ms between clients and servers, where each client performs a ping test to a server. During this test, we intercept the network traffic and apply different latency values for each client, applying 10, 30, 50, and 100 ms, respectively. Annex II shows the custom program that applies the correct latency to a specific client.

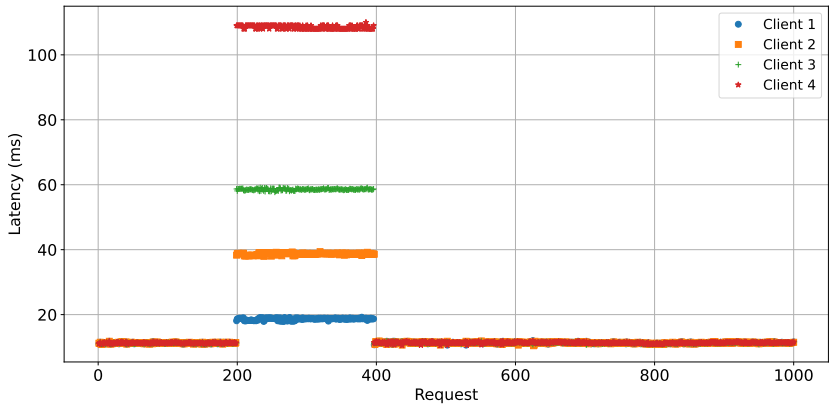


Figure 4.22: Latency per Request per Client

Fig. 4.22 shows the latency observed for each request of a given client. This example shows how the user can enhance a given experiment of applying latency in a more focused approach, allowing the creation of new testing scenarios with a simple mechanism that other network emulators such as Kollaps do not provide.

4.5 Summary

In this chapter, we present the evaluation done to validate our network emulator. We detailed our evaluation plan, specifying the objectives for our testing and the tools used for those efforts. Next, we presented the tests to evaluate the objectives of our evaluation plan, validating our network emulator regarding the emulated network properties and performance.

In addition to network properties, we also evaluated the available network operations the user can leverage for an experiment, demonstrating its impact in the tools used.

To finish this chapter, we presented novel operations, that allow the user to retrieve information about the network traffic or directly influence it, enabling the user to further extend the capabilities of our network emulator to create new and improved testing scenarios.

The next chapter concludes this thesis, presenting improvements and new features to complement this network emulator as future work.

5

Conclusion

With the increased interest in microservice architectures, distributed systems, and cloud adoption, it is become clear the importance of network infrastructure to provide a quick response to the clients. As such, to ensure that distributed systems operate correctly, it is necessary the relevant testing to ensure smooth operation, when face with unpredictable network conditions.

In this thesis, we have studied the existing approaches to allow developers to deploy and test their systems, and learned their drawbacks. We learned about existing tools that allow traffic manipulation in order to perform network emulation and how current state-of-the-art network emulators leverage some of these tools. This lead us to create our network emulator, GONE, a network emulator capable of creating an interactive network where the users can deploy their applications.

Our network emulator, leverages Docker to allow the user to create deploy their own images without having to perform any code modifications to accommodate our network emulator and resorts to using eBPF programs and XDP sockets to intercept network traffic between containers and insert it into an emulated network. This allows the user to configure their desired network properties, provide an interactive network, which the user modifies it during runtime. Due to the design of our network emulator, the user can obtain direct access to the network traffic, allowing the ability to create custom programs, further extending the capabilities of our network emulator. This simplified approach to network traffic access allows the creation of new testing environments that no other network emulator is capable of providing.

The evaluation conducted in this thesis leads us to conclude that our network emulator is capable of providing similar network environments compared to other network emulators, and in some cases supporting larger network topologies, due to not being restricted to the limitations of TC, while also providing mechanisms to further improve the capabilities of our network emulator.

Future Work

In this section, we present possible future work that improves our current solution. These improvements are as follows:

Communication through shared memory: Internally, our network emulator receives the network traffic through unix sockets, requiring encoding and decoding of messages, leading to extra processing. By communicating through shared memory, its reduced the number of copy operations done by the proxy and the network emulator, improving performance.

Better Docker integration: Currently, our network emulator issues commands through the Docker-CLI. This approach however is quite limited since the network emulator is not aware of what happens in the docker environment, and cannot detect when containers fail.

Unidirectional properties: When configuring the network properties of a network link, the user configures the network properties for both directions of the link, which proves troublesome when is necessary to only observe the network properties in a single direction.

Storing experiments: Implementing a mechanism to store the current running network into a format that could be read, restoring the emulation.

Dashboard: Implementing a dashboard to observe the current network.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[2] *Enterprises using cloud computing*. Accessed Fev. 2023. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises#Enterprises.E2.80.99_dependence_on_cloud_computing (cit. on p. 1).

[3] C. Guan, D. Ding, and J. Guo. "Web3.0: A Review And Research Agenda". In: *2022 RIVF International Conference on Computing and Communication Technologies (RIVF)*. 2022, pp. 653–658. DOI: 10.1109/RIVF55975.2022.10013794 (cit. on p. 1).

[4] *Emulab*. Accessed Jan. 2023. URL: https://www.emulab.net/portal/frontpage.php (cit. on pp. 1, 2, 7).

[5] *PlanetLab*. Accessed Jan. 2023. URL: https://planetlab.cs.princeton.edu/tutorial.html (cit. on pp. 1, 2, 7).

[6] *Cloud computing services: Microsoft Azure*. Accessed Jan. 2023. URL: https://azure.microsoft.com/ (cit. on pp. 1, 2, 7).

[7] *Google Cloud Platform*. Accessed Jan. 2023. URL: https://cloud.google.com/ (cit. on pp. 1, 2, 7).

[8] *cloud computing services - amazon web services*. Accessed Jan. 2023. URL: https://signin.aws.amazon.com/ (cit. on pp. 1, 2, 7).

[9] Nsnam. *NS-3 Network Simulator*. URL: https://www.nsnam.org/ (cit. on pp. 2, 7, 9, 21).

[10] A. Varga and R. Hornig. "An overview of the OMNeT++ simulation environment". In: 2008, p. 60. DOI: 10.1145/1416222.1416290 (cit. on pp. 2, 7, 9, 20).

[11] R. Jansen and N. Hopper. "Shadow: Running Tor in a Box for Accurate and Efficient Experimentation". In: *Symposium on Network and Distributed System Security*. See also https://shadow.github.io. 2012 (cit. on pp. 2, 22).

[12] K. Kaur, J. Singh, and N. Ghumman. "Mininet as Software Defined Networking Testing Platform". In: 2014 (cit. on pp. 2, 9, 17, 25, 28).

[13] P. Gouveia et al. "Kollaps: Decentralized and Dynamic Topology Emulation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387540 (cit. on pp. 2, 7–9, 26, 28).

[14] A. de Oliveira Atalaia et al. "SYMPHONY: A SCALABLE EMULATOR FOR DISTRIBUTED SYSTEMS". In: 2023. URL: https://api.semanticscholar.org/CorpusID:260061478 (cit. on pp. 2, 27, 28).

[15] L. S. Vailshery. *Challenges surrounding SAAS apps worldwide 2021*. Accessed Fev. 2023. URL: https://www.statista.com/statistics/1267901/saas-apps-biggest-challenges-organizations-worldwide/ (cit. on p. 7).

[16] Accessed Jan. 2023. URL: https://infrastructuremap.microsoft.com/ (cit. on p. 7).

[17] L. Rizzo. "Dummynet: A Simple Approach to the Evaluation of Network Protocols". In: 27.1 (1997), 31–41. ISSN: 0146-4833. DOI: 10.1145/251007.251012 (cit. on p. 7).

[18] A. Vahdat et al. "Scalability and Accuracy in a Large-Scale Network Emulator". In: *SIGOPS Oper. Syst. Rev.* 36.SI (2003), 271–284. ISSN: 0163-5980. DOI: 10.1145/844128.844154 (cit. on pp. 7, 24, 28).

[19] J. Banks et al. *Discrete-event System Simulation*. Prentice-Hall international series in industrial and systems engineering. Pearson Prentice Hall, 2005. ISBN: 9780131446793. URL: https://books.google.pt/books?id=CWZRAAAAMAAJ (cit. on p. 8).

[20] D. Dugaev and E. Siemens. "A Wireless Mesh Network NS-3 Simulation Model: Implementation and Performance Comparison With a Real Test-Bed". In: 2019. DOI: 10.13142/kt10002.01 (cit. on pp. 8, 21).

[21] G. Carneiro, H. Fontes, and M. Ricardo. "Fast prototyping of network protocols through ns-3 simulation model reuse". In: *Simulation Modelling Practice and Theory* 19 (2011-10), pp. 2063–2075. DOI: 10.1016/j.simpat.2011.06.002 (cit. on pp. 8, 21).

[22] R. Chertov, S. Fahmy, and N. B. Shroff. "Fidelity of Network Simulation and Emulation: A Case Study of TCP-Targeted Denial of Service Attacks". In: *ACM Trans. Model. Comput. Simul.* 19.1 (2009). ISSN: 1049-3301. DOI: 10.1145/1456645.1456649 (cit. on p. 8).

[23] E. Lochin, T. Pérennou, and L. Dairaine. "When should I use network emulation?" In: *annals of telecommunications - annales des télécommunications* 67.5 (2012), pp. 247–255. ISSN: 1958-9395. DOI: 10.1007/s12243-011-0268-5 (cit. on pp. 8, 9).

[24] P. Bailis and K. Kingsbury. "The Network is Reliable: An Informal Survey of Real-World Communications Failures". In: *Queue* 12.7 (2014), 20–32. ISSN: 1542-7730. DOI: 10.1145/2639988.2655736 (cit. on p. 8).

[25] J. Postel. *Transmission control protocol*. Accessed Jan. 2023. URL: https://www.rfc-editor.org/rfc/rfc793 (cit. on p. 9).

[26] M. A. To, M. Cano, and P. Biba. "DOCKEMU – A Network Emulation Tool". In: *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*. 2015, pp. 593–598. DOI: 10.1109/WAINA.2015.107 (cit. on p. 9).

[27] *Accelerated, containerized application development*. Accessed Jan. 2023. URL: https://www.docker.com/ (cit. on pp. 9, 18, 19, 27, 28).

[28] S. Hemminger. "Network emulation with NetEm". In: *Linux Conf Au* (2005) (cit. on pp. 9, 10).

[29] P. Wette et al. "MaxiNet: Distributed emulation of software-defined networks". In: *2014 IFIP Networking Conference*. 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857078 (cit. on pp. 9, 17, 26).

[30] M. A. M. Vieira et al. "Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications". In: 53.1 (2020). ISSN: 0360-0300. DOI: 10.1145/3371038 (cit. on p. 10).

[31] S. McCanne and V. Jacobson. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. USENIX Association, 1993, p. 2 (cit. on p. 10).

[32] *Home | TCPDUMP; LIBPCAP — tcpdump.org*. Accessed 06-02-2025. URL: https://www.tcpdump.org/ (cit. on pp. 10, 36).

[33] *BPF: Introduce bpf_tail_call() helper*. URL: https://lwn.net/Articles/645169/ (cit. on p. 10).

[34] S. Becker et al. *Network Emulation in Large-Scale Virtual Edge Testbeds: A Note of Caution and the Way Forward*. 2022. DOI: 10.48550/ARXIV.2208.05862 (cit. on p. 10).

[35] M. Abranches et al. "Efficient Network Monitoring Applications in the Kernel with eBPF and XDP". In: *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2021, pp. 28–34. DOI: 10.1109/NFV-SDN53031.2021.9665095 (cit. on p. 10).

[36] C. Liu et al. "A protocol-independent container network observability analysis system based on eBPF". In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. 2020, pp. 697–702. DOI: 10.1109/ICPADS51040.2020.00099 (cit. on p. 10).

[37] J. Levin. "ViperProbe: Using eBPF Metrics to Improve Microservice Observability". In: 2020 (cit. on p. 10).

[38] T. Høiland-Jørgensen et al. "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel". In: CoNEXT '18. Association for Computing Machinery, 2018, 54–66. ISBN: 9781450360807. DOI: 10.1145/3281411.3281443 (cit. on pp. 11, 12).

[39] B. T. Magnus Karlsson. *The Path to DPDK Speeds for AF-XDP*. https://lpc.events/event/2/contribution Accessed 07-02-2025. 2018 (cit. on pp. 11, 12).

[40] *GitHub - xdp-project/xdp-tutorial: XDP tutorial — github.com*. Accessed 06-02-2025. URL: https://github.com/xdp-project/xdp-tutorial (cit. on pp. 11, 81).

[41] J. Corbet. *Accelerating networking with AF_XDP [lwn.net] — lwn.net*. https://lwn.net/Articles/750845/ Accessed 07-02-2025. 2018 (cit. on p. 12).

[42] Accessed Jan. 2023. URL: https://almesberger.net/cv/papers/tcio8.pdf (cit. on p. 13).

[43] *1. introduction to linux traffic control*. Accessed Jan. 2023. URL: https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html (cit. on p. 13).

[44] Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man8/tc.8.html (cit. on pp. 13, 26, 27).

[45] *The netfilter.org project*. Accessed Jan. 2023. URL: https://www.netfilter.org/ (cit. on p. 14).

[46] A. Jones. *Netfilter and IPTables: A Structural Examination*. 2004 (cit. on p. 14).

[47] Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man8/arptables-nft.8.html (cit. on p. 14).

[48] Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man8/iptables.8.html (cit. on p. 14).

[49] Accessed Jan. 2023. URL: https://linux.die.net/man/8/ip6tables (cit. on p. 14).

[50] *NFTABLES: A new packet filtering engine*. Accessed Jan. 2023. URL: https://lwn.net/Articles/324989/ (cit. on p. 14).

[51] *The return of nftables*. Accessed Jan. 2023. URL: https://lwn.net/Articles/564095/ (cit. on p. 14).

[52] *Nftables Wiki page*. Accessed Jan. 2023. URL: https://wiki.nftables.org/wiki-nftables/index.php/Main_Page (cit. on pp. 14, 15).

[53] D. Plummer. *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826 (Internet Standard). RFC. Updated by RFCs 5227, 5494. Fremont, CA, USA: RFC Editor, 1982-11. DOI: 10.17487/RFC0826 (cit. on p. 14).

[54] J. Postel. *Internet Protocol*. RFC 791 (Internet Standard). RFC. Updated by RFCs 1349, 2474, 6864. Fremont, CA, USA: RFC Editor, 1981-09. DOI: 10.17487/RFC0791 (cit. on p. 14).

[55] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). RFC. Obsoleted by RFC 8200, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. Fremont, CA, USA: RFC Editor, 1998-12. DOI: 10.17487/RFC2460 (cit. on p. 14).

[56] M. Boye. "Netfilter Connection Tracking and NAT Implementation". In: 2012 (cit. on p. 15).

[57] P. Sutter. *Benchmarking nftables*. Accessed Jan. 2023. URL: https://developers.redhat.com/blog/2017/04/11/benchmarking-nftables#the_first_test (cit. on p. 16).

[58] A. Tanenbaum, D. Wetherall, and N. Feamster. "Chapter 5 - The Network Layer". In: *Computer Networks, EBook, Global Edition*. 6th ed. Pearson Education, Limited, 2021, 435–441 (cit. on p. 16).

[59] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: 10.1109/JPROC.2014.2371999 (cit. on p. 16).

[60] P. Goransson, T. Culver, and C. Black. "Chapter 2 Why SDN?" In: *Software defined networks: A comprehensive approach*. Morgan Kaufmann Publishers, 2017, 21–35 (cit. on p. 16).

[61] N. McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (2008), 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746 (cit. on pp. 17, 25).

[62] *Production quality, Multilayer Open Virtual Switch*. Accessed Jan. 2023. URL: https://www.openvswitch.org/ (cit. on p. 17).

[63] M. Portnoy. *Virtualization Essentials*. 1st. USA: SYBEX Inc., 2012. ISBN: 1118176715 (cit. on p. 18).

[64] *Container and virtualization tools*. Accessed Jan. 2023. URL: https://linuxcontainers.org/ (cit. on pp. 18, 19).

[65] *cgroups(7) — Linux manual page*. Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html (cit. on pp. 18, 26).

[66] *namespaces(7) — Linux manual page*. Accessed Jan. 2023. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (cit. on pp. 18, 26).

[67] J "Berggren and J. Karlsson. ""Differences in performance between containerization & virtualization : With a focus on HTTP requests"". In: *"Dissertation"* () (cit. on p. 19).

[68] B. Bashari Rad, H. Bhatti, and M. Ahmadi. "An Introduction to Docker and Analysis of its Performance". In: *IJCSNS International Journal of Computer Science and Network Security* 173 (2017-03), p. 8 (cit. on p. 19).

[69] S. Soltesz et al. "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors". In: *SIGOPS Oper. Syst. Rev.* 41.3 (2007), 275–287. ISSN: 0163-5980. DOI: 10.1145/1272998.1273025 (cit. on p. 19).

[70] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (2010), 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922 (cit. on pp. 19, 64).

[71] S. Shirinbab, L. Lundberg, and E. Casalicchio. "Performance evaluation of containers and virtual machines when running Cassandra workload concurrently". In: *Concurrency and Computation: Practice and Experience* 32.17 (2020), e5693. DOI: https://doi.org/10.1002/cpe.5693 (cit. on p. 19).

[72] *Production-grade container orchestration*. Accessed Jan. 2023. URL: https://kubernetes.io/ (cit. on pp. 19, 27, 28).

[73] *Swarm mode overview*. Accessed Fev. 2023. 2023. URL: https://docs.docker.com (cit. on p. 19).

[74] N. Marathe, A. Gandhi, and J. M. Shah. "Docker Swarm and Kubernetes in Cloud Computing Environment". In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. 2019, pp. 179–184. DOI: 10.1109/ICOEI.2019.8862654 (cit. on p. 19).

[75] I. M. A. Jawarneh et al. "Container Orchestration Engines: A Thorough Functional and Performance Comparison". In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053 (cit. on p. 19).

[76] A. Varga. *Simulation Manual - Version 6.0*. Accessed Jan. 2023. URL: https://doc.omnetpp.org/omnetpp/manual/index.html (cit. on p. 20).

[77] K. Wehrle, M. Güneş, and J. Gross, eds. *Modeling and Tools for Network Simulation*. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-12331-3 (cit. on pp. 20, 21).

[78] F. Estevez et al. "Enabling Validation of IEEE 802.15.4 Performance through a New Dual-Radio Omnet++ Model". In: *Elektronika ir Elektrotechnika* 22 (2016). DOI: 10.5755/j01.eie.22.3.15321 (cit. on p. 21).

[79] M. Stoffers et al. *Enabling Distributed Simulation of OMNeT++ INET Models*. 2014. DOI: 10.48550/ARXIV.1409.0994 (cit. on p. 21).

[80] J. Oladipo, M. C. duPlessis, and T. B. Gibbon. "Implementation and validation of an Omnet++ optical burst switching simulator". In: *2017 International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*. 2017, pp. 1–6. DOI: 10.23919/PEMWN.2017.8308024 (cit. on p. 21).

[81] L. Kamarudin et al. "Review and Modeling of Vegetation Propagation Model for Wireless Sensor Networks Using Omnet++". In: *Network Applications, Protocols and Services, International Conference on* 0 (2010-09), pp. 78–83. DOI: 10.1109/NETAPPS.2010.21 (cit. on p. 21).

[82] N. S. Prashanth. "OpenFlow Switching Performance using Network Simulator - 3". In: *Dissertation* (2016) (cit. on p. 21).

[83] L. Alberro et al. "Experimenting with Routing Protocols in the Data Center: An ns-3 Simulation Approach". In: *Future Internet* 14.10 (2022). ISSN: 1999-5903. DOI: 10.3390/fi14100292 (cit. on p. 21).

[84] A. Montresor and M. Jelasity. "PeerSim: A scalable P2P simulator". In: *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. 2009, pp. 99–100. DOI: 10.1109/P2P.2009.5284506 (cit. on p. 21).

[85] I. Kazmi and S. F. Y. Bukhari. "PeerSim: An Efficient & Scalable Testbed for Heterogeneous Cluster-based P2P Network Protocols". In: *2011 UkSim 13th International Conference on Computer Modelling and Simulation*. 2011, pp. 420–425. DOI: 10.1109/UKSIM.2011.86 (cit. on p. 21).

[86] *PeerSim: A peer-to-peer simulator*. Accessed Fev. 2023. URL: https://peersim.sourceforge.net/ (cit. on p. 21).

[87] J. Leitão, J. Pereira, and L. Rodrigues. "HyParView: a membership protocol for reliable gossip-based broadcast". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Edinburgh, UK, 2007, pp. 419–429 (cit. on p. 22).

[88] J. Leitão. "Gossip-Based Broadcast Protocols". MA thesis. Faculdade de Ciências da Universidade de Lisboa, 2007 (cit. on p. 22).

[89] R. Jansen, J. Newsome, and R. Wails. "Co-opting Linux Processes for High-Performance Network Simulation". In: *USENIX Annual Technical Conference*. See also https://netsim-atc2022.github.io. 2022 (cit. on p. 22).

[90] F. Shirazi, C. Diaz, and J. Wright. "Towards Measuring Resilience in Anonymous Communication Networks". In: *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society*. WPES '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 95–99. ISBN: 9781450338202. DOI: 10.1145/2808138.2808152 (cit. on p. 22).

[91] *The Tor Project | Privacy & Freedom Online — torproject.org*. https://www.torproject.org/. Accessed 22-02-2025 (cit. on p. 22).

[92] *Shadow Experiments for Congestion Control (#40404) · Issues · The Tor Project / Core / Tor · GitLab — gitlab.torproject.org*. https://gitlab.torproject.org/tpo/core/tor/-/issues/40404. Accessed 22-02-2025 (cit. on p. 22).

[93] *veth(4) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man4/veth.4.html. Accessed 25-02-2025 (cit. on p. 26).

[94] Accessed Jan. 2023. URL: https://community.cloudflare.com/t/learning-center-what-is-gre-tunneling-how-gre-protocol-works/347586 (cit. on p. 26).

[95] M. Peuster, H. Karl, and S. van Rossem. "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490 (cit. on p. 26).

[96] S. Amaro. MA thesis. 2021. URL: https://fenix.tecnico.ulisboa.pt/cursos/meic-t/dissertacao/1972678479055152 (cit. on p. 26).

[97] *GitHub - David-Antunes/gone — github.com*. https://github.com/David-Antunes/gone. Accessed 04-03-2025 (cit. on p. 31).

[98] *GitHub - David-Antunes/gone-proxy — github.com*. https://github.com/David-Antunes/gone-proxy. Accessed 04-03-2025 (cit. on p. 31).

[99] *GitHub - David-Antunes/gone-rtt — github.com*. https://github.com/David-Antunes/gone-rtt. Accessed 04-03-2025 (cit. on p. 31).

[100] *Neo4j Graph Database & Analytics – The Leader in Graph Databases — neo4j.com*. https://neo4j.com/. Acessed 04-03-2025 (cit. on p. 32).

[101] *GitHub - David-Antunes/gone-cli — github.com*. https://github.com/David-Antunes/gone-cli. Accessed 04-03-2025 (cit. on p. 32).

[102] *GitHub - David-Antunes/gone-agent — github.com*. https://github.com/David-Antunes/gone-agent. Accessed 04-03-2025 (cit. on p. 32).

[103] *The Go Programming Language — go.dev*. https://go.dev/src/runtime/HACKING. [Accessed 13-07-2025] (cit. on p. 38).

[104] J. Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: (2014). DOI: 10.48550/ARXIV.1407.3561 (cit. on p. 59).

[105] *GitHub - brianfrankcooper/YCSB: Yahoo! Cloud Serving Benchmark — github.com*. https://github.com/brianfrankcooper/YCSB. [Accessed 02-04-2025] (cit. on p. 64).

I

eBPF Program Example

Fig. I.1

```
      /* SPDX-License-Identifier: GPL-2.0 */

#include <linux/bpf.h>

#include <bpf/bpf_helpers.h>

struct {
  __uint(type, BPF_MAP_TYPE_XSKMAP);
  __type(key, __u32);
  __type(value, __u32);
  __uint(max_entries, 64);
} xsks_map SEC(".maps");

struct {
  __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
  __type(key, __u32);
  __type(value, __u32);
  __uint(max_entries, 64);
} xdp_stats_map SEC(".maps");

SEC("xdp")
int xdp_sock_prog(struct xdp_md *ctx)
{
  int index = ctx->rx_queue_index;
  __u32 *pkt_count;

  pkt_count = bpf_map_lookup_elem(&xdp_stats_map, &index);
  if (pkt_count) {

    /* We pass every other packet */
    if ((*pkt_count)++ & 1)
    return XDP_PASS;
  }

  /* A set entry here means that the correspnding queue_id
   * has an active AF_XDP socket bound to it. */
  if (bpf_map_lookup_elem(&xsks_map, &index))
    return bpf_redirect_map(&xsks_map, index, 0);

  return XDP_PASS;
}

char _license[] SEC("license") = "GPL";
```

Figure I.1: Example code to redirect network packets to an AF_XDP socket. Extracted from [40]

II

GONE EXTERNAL PROGRAMS

Fig. II.2 shows an example of a GO program that receives sniffing traffic from a unix socket, and converts it to the PCAP format.

Listing II.1: Example go program to apply latency

```
package main

import (
    "encoding/gob"
    "fmt"
    "net"
    "os"
    "time"

    "github.com/David-Antunes/gone-proxy/xdp"
    "github.com/google/gopacket"
    "github.com/google/gopacket/layers"
)

func main() {

    if len(os.Args) == 1 {
        fmt.Println("missing socket id")
        return
    }
    if len(os.Args) > 2 {
        fmt.Println("too many arguments")
        return
    }
    conn, err := net.Dial("unix", os.Args[1])

    if err != nil {
        panic(err)
    }
    dec := gob.NewDecoder(conn)
    enc := gob.NewEncoder(conn)
```

```
"/addNode" --- Adds a new node to the emulation
"/addBridge" --- Registers a new bridge
"/addRouter" --- Registers a new Router

"/connectNodeToBridge" --- Connects a node to a bridge
"/connectBridgeToRouter" --- Connects a bridge to a router
"/connectRouterToRouter" --- Connects two routers

"/inspectNode" --- Obtains information about a node
"/inspectBridge" --- Obtains information about a bridge
"/inspectRouter" --- Obtains information about a router

"/removeNode" --- Removes a node
"/removeBridge" --- Removes a bridge
"/removeRouter" --- Removes a router

"/disconnectNode" --- Disconnects a node from its bridge
"/disconnectBridge" --- Disconnects a bridge from its router
"/disconnectRouters" --- Disconnects two routers

"/forget" --- Clears all routing rules from a router
"/propagate" --- Propagates the routes of a router

"/sniffNode" --- Sniffs the network traffic from a node
"/sniffBridge" --- Sniffs the network traffic from a bridge
"/sniffRouters" --- Sniffs the network traffic between two routers
"/stopSniff" --- Stops sniffing a link
"/listSniffers" --- Lists all active sniffing links

"/interceptNode" --- Intercepts the network traffic between a node and its bridge
"/interceptBridge" --- Intercepts the network traffic between a bridge and its router
"/interceptRouter" --- Intercepts the network traffic between routers
"/stopIntercept" --- Stops intercepting a link
"/listIntercepts" --- Lists all active intercepted links

"/disruptNode" --- Disables the link between a node and its bridge
"/stopDisruptNode" --- Stops disabling the link of a node and its bridge
"/disruptBridge" --- Disables the link between a bridge and its router
"/stopDisruptBridge" --- Stops disabling the link of a bridge and its router
"/disruptRouters" --- Disables the link between routers
"/stopDisruptRouters" --- Stops disabling the link between routers

"/startBridge" --- Starts a stopped bridge
"/stopBridge" --- Stops a bridge

"/startRouter" --- Starts a stopped router
"/stopRouter" --- Stops a router

"/pause" --- Pauses a node
"/unpause" --- Unpauses a node
```

Figure II.1: Gone-api available endpoints

```
package main

import (
	"encoding/gob"
	"fmt"
	"net"
	"os"

	"github.com/David-Antunes/gone-proxy/xdp"
	"github.com/google/gopacket"
	"github.com/google/gopacket/layers"
	"github.com/google/gopacket/pcapgo"
)
func main() {
	if len(os.Args) == 1 {
		fmt.Println("missing socket id")
		return
	}
	if len(os.Args) > 2 {
		fmt.Println("too many arguments")
		return
	}
	conn, err := net.Dial("unix", os.Args[1])

	if err != nil {
		panic(err)
	}
	dec := gob.NewDecoder(conn)

	w := pcapgo.NewWriter(os.Stdout)
	w.WriteFileHeader(uint32(65535), layers.LinkTypeEthernet)
	for {
		var frame *xdp.Frame

		err := dec.Decode(&frame)
		if err != nil {
			panic(err)
		}
		packet := gopacket.NewPacket(frame.FramePointer, layers.LinkTypeEthernet,
		    gopacket.Default)
		w.WritePacket(gopacket.CaptureInfo{Timestamp: frame.Time, CaptureLength:
		    frame.FrameSize, Length: frame.FrameSize, InterfaceIndex: 1}, packet.
		    Data())
	}
}
```

Figure II.2: Example go program to convert the network traffic into pcap format

```
channel := make(chan *xdp.Frame, 1000)

        for {
            frame := <-channel

            err := enc.Encode(&frame)
            if err != nil {
                panic(err)
            }
        }
    }()

    for {
        var frame *xdp.Frame
        err := dec.Decode(&frame)
        if err != nil {
            panic(err)
        }
        packet := gopacket.NewPacket(frame.FramePointer, layers.LinkTypeEthernet,
             gopacket.NoCopy)

        if ipLayer := packet.Layer(layers.LayerTypeIPv4); ipLayer != nil {
            ip := ipLayer.(*layers.IPv4)

            if ip.SrcIP.String() == "10.1.0.101" {
                go func() {
                    time.Sleep(10 * time.Millisecond)
                    channel <- frame
                }()
            } else {
                channel <- frame
            }
        } else {
            channel <- frame
        }
    }
}
```

Fig. II shows a custom GO program that receives intercepted network traffic. For each network packet received, it reads its source IP address, and applies a latency of 10 milliseconds.

Listing II.2: External Program that applies 10% packet loss

```
package main

import (
        "encoding/gob"
        "fmt"
        "math/rand/v2"
        "net"
```

```
	"os"

	"github.com/David-Antunes/gone-proxy/xdp"
)

func main() {
	if len(os.Args) == 1 {
		fmt.Println("missing socket id")
		return
	}
	if len(os.Args) > 2 {
		fmt.Println("too many arguments")
		return
	}

	conn, err := net.Dial("unix", os.Args[1])

	if err != nil {
		panic(err)
	}
	// Encoders and decoders for packets sent by the emulation
	dec := gob.NewDecoder(conn)
	enc := gob.NewEncoder(conn)

	for {
		var frame *xdp.Frame
		err := dec.Decode(&frame)
		if err != nil {
			panic(err)
		}
		if rand.Float64() <= 0.10 {
			continue
		}
		err = enc.Encode(&frame)
		if err != nil {
			panic(err)
		}
	}
}
```

Fig. II is a custom program that applies a drop rate of 10%.

Listing II.3: External Program that applies latency to a specific IP address

```
package main

import (
	"encoding/gob"
	"fmt"
	"net"
	"os"
```

```
	"time"

	"github.com/David-Antunes/gone-proxy/xdp"
	"github.com/google/gopacket"
	"github.com/google/gopacket/layers"
)

func main() {

	if len(os.Args) == 1 {
		fmt.Println("missing socket id")
		return
	}

	if len(os.Args) > 2 {
		fmt.Println("too many arguments")
		return
	}

	conn, err := net.Dial("unix", os.Args[1])

	if err != nil {
		panic(err)
	}
	// Encoders and decoders for packets sent by the emulation
	dec := gob.NewDecoder(conn)
	enc := gob.NewEncoder(conn)

	// Routine to send packet back to the emulation
	channel := make(chan *xdp.Frame, 10000)
	go func() {
		for {
			frame := <-channel

			err := enc.Encode(&frame)
			if err != nil {
				panic(err)
			}
		}
	}()

	// Main routine to distribute the load across the clients
 timer := time.Now().Add(10*time.Second)
 stop := false

 go func() {
	for !stop {
		var frame *xdp.Frame
		err := dec.Decode(&frame)
		if err != nil {
```

```
                panic(err)
            }
    go func() {

            packet := gopacket.NewPacket(frame.FramePointer, layers.LinkTypeEthernet,
                gopacket.NoCopy)

            if ipLayer := packet.Layer(layers.LayerTypeIPv4); ipLayer != nil {
                ip := ipLayer.(*layers.IPv4)

                switch ip.SrcIP.String() {
                case "10.1.0.100":
                    time.Sleep(time.Until(frame.Time.Add(10 * time.Millisecond)))
                case "10.1.0.101":
                    time.Sleep(time.Until(frame.Time.Add(30 * time.Millisecond)))
                case "10.1.0.102":
                    time.Sleep(time.Until(frame.Time.Add(50 * time.Millisecond)))
                case "10.1.0.103":
                    time.Sleep(time.Until(frame.Time.Add(100 * time.Millisecond)))
                default:
                    fmt.Println("failed")
                }
            }
            channel <- frame
  }()
       }
}()
  time.Sleep(time.Until(timer))
  stop = true

  for {
            var frame *xdp.Frame
            err := dec.Decode(&frame)
            if err != nil {
                panic(err)
            }
    channel <- frame
  }
}
```

Fig. II is a custom program that applies custom latency depending on the source IP address.